Maria Paola Bonacina (Ed.)

LNAI 7898

Automated Deduction – CADE-24

24th International Conference on Automated Deduction
Lake Placid, NY, USA, June 2013
Proceedings

Lecture Notes in Artificial Intelligence 7898

Subseries of Lecture Notes in Computer Science

Maria Paola Bonacina (Ed.)

Automated Deduction – CADE-24

24th International Conference on Automated Deduction
Lake Placid, NY, USA, June 9-14, 2013
Proceedings

Springer

Volume Editor

Maria Paola Bonacina
Università degli Studi di Verona
Dipartimento di Informatica
Strada Le Grazie 15, 37134 Verona, Italy
E-mail: mariapaola.bonacina@univr.it

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-38573-5 e-ISBN 978-3-642-38574-2
DOI 10.1007/978-3-642-38574-2
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013938607

CR Subject Classification (1998): I.2.3, I.2, F.4.1, F.3, F.4, D.2.4, D.3

LNCS Sublibrary: SL 7 – Artificial Intelligence

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the papers presented at the 24th International Conference on Automated Deduction (CADE-24), held during June 9–14, 2013, in Lake Placid, New York, USA. CADE is the major forum for the presentation of research in all aspects of automated deduction, including foundations, applications, implementations, and practical experiences.

The Program Committee accepted 31 papers (22 full papers and 9 system descriptions) out of 71 submissions (53 full papers and 18 system descriptions). The acceptance rate was 43.66% overall, 41.51% for full papers, and 50% for system descriptions. Each submission was reviewed by at least three Program Committee (PC) members or external reviewers appointed by the PC members in charge. The main criteria for evaluation were originality and significance, technical quality and completeness, comparison with related work and completeness of references, quality of presentation, clarity, and readability.

The Best Paper Award was conferred to Radu Iosif (Verimag and CNRS, Grenoble, France), Adam Rogalewicz, and Jiri Simacek (Brno University of Technology, Czech Republic), for their paper entitled "The Tree Width of Separation Logic with Recursive Definitions," which proves decidability of satisfiability and entailment in an expressive fragment of separation logic, a logic relevant to program verification. According to separation logic experts who reviewed it, this paper closes in a creative and insightful way a problem that was open since 2004 and that was attacked unsuccessfully by several scholars.

The technical program of the conference included four invited talks by Jean-Christophe Filliâtre (CNRS and LRI Université Paris Sud XI, France), on "One Logic to Use Them All," Greg Morrisett (Harvard University, USA) on "Defining, Testing, and Reasoning About an x86 Decoder," Natarajan Shankar (SRI International, USA) on "Automated Reasoning, Fast and Slow," and Douglas R. Smith (Kestrel Institute and Kestrel Technology LLC, USA) on "Coalgebraic Specification and Refinement." This volume includes the invited papers by Jean-Christophe Filliâtre and Natarajan Shankar. The talk by Doug Smith focused on using coalgebraic concepts to specify the requirements on dynamical systems and then use deductive techniques to calculate refinements of the specifications into correct-by-construction code. The emphasis was on deduction for purposes of generating correct code rather than performing ad hoc verification on manually written code.

During the conference, the Herbrand Award for Distinguished Contributions to Automated Reasoning was presented to Greg Nelson for his invention of equality sharing, also known as the Nelson-Oppen method, and his pioneering work on theorem proving and program checking, including fast congruence closure algorithms and the Simplify theorem prover. The Selection Committee for the

Herbrand Award consisted of the CADE-24 Program Committee members, the trustees of CADE Inc., and the Herbrand Award winners of the last ten years.

The conference issued a call for workshops out of which the following seven proposals were approved:

- Automated Deduction: Decidability, Complexity, Tractability (ADDCT) by Silvio Ghilardi, Ulrike Sattler, Viorica Sofronie-Stokkermans, and Ashish Tiwari
- Automated Reasoning in Security (ARSEC) by Paliath Narendran, Christopher A. Lynch, Andrew Marshall, and Dan Dougherty
- Empirically Successful Automated Reasoning with Artificial Intelligence (ESARAI) by Boris Konev, Stephan Schulz, and Geoff Sutcliffe
- Knowledge-Intensive Automated Reasoning (KInAR) by Ulrich Furbach and Björn Pelzer
- Methods for Modalities (M4M) by Carlos Areces
- Proof Exchange for Theorem Proving (PxTP) by Jasmin Christian Blanchette and Josef Urban
- The StarExec Web Service for the Evaluation of Logic Solvers (StarExec) by Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli

Similarly, a call for tutorials generated the following five:

- Reasoning in Lightweight Description Logics, by Franz Baader
- Program Verification with the KeY System, by Bernhard Beckert and Reiner Hähnle
- Becoming a Power User of SMT: The CVC4 Solver, by Morgan Deters, Dejan Jovanović, Clark W. Barrett, and Cesare Tinelli
- State-of-the-art SAT Solving, by Marijn Heule
- The Twelf System, by Carsten Schürmann, Taus Brock-Nannestad, and Chris Martens

During the conference, the CADE-24 ATP System Competition – CASC-24 – was held, organized by Geoff Sutcliffe, who contributed the following description.

The CADE ATP System Competition (CASC) is an annual evaluation of fully automatic, classical logic automated theorem proving (ATP) systems – the world championship for such systems. Its main purpose is to provide a public evaluation of the relative capabilities of ATP systems. Additionally, CASC aims at stimulating ATP research, motivating development and implementation of robust, useful and easily deployable ATP systems, providing an inspiring environment for personal interaction between ATP researchers, and exposing ATP systems within and beyond the ATP community. Fulfillment of these objectives offers insight and stimulus for the development of more powerful ATP systems, leading to increased and more effective use. The CASC-24 website provides access to all systems and competition resources: `http://www.tptp.org/CASC/24`. CASC-24 was run in divisions according to problem and system characteristics:

- THF: Typed Higher-Order Form Theorems (axioms with a provable conjecture)
- TFA: Typed First-Order with Arithmetic Theorems (axioms with a provable conjecture)
- FOF: First-Order Form Theorems (axioms with a provable conjecture)
- FNT: First-Order Form Syntactically Non-propositional Non-theorems
- EPR: Effectively PRopositional Clause Normal Form Theorems and Non-theorems
- LTB: First-Order Form Theorems (axioms with a provable conjecture) from Large Theories, presented in Batches

In the THF, TFA, and EPR divisions, provers were ranked based on the number of problems solved, without being required to produce a proof or model. In the FOF, FNT, and LTB divisions, provers were ranked based on the number of problems solved with an acceptable proof or model output. The LTB division featured a 24-hour training period before the competition, during which systems could use a set of problems and solutions for tuning and training, but no human intervention was allowed. Problems for CASC-24 were taken from the TPTP Problem Library, using a version released after the start of the competition, so that new problems had not been seen by the entrants. Ties were broken according to the average time over problems solved.

Several students received Woody Bledsoe Travel Awards, thus named to remember the late Woody Bledsoe, funded by CADE Inc. to sponsor student participation. Best Paper Award winners, CASC divisions winners, and Woody Bledsoe Travel Awards recipients were announced during the banquet.

Many people contributed to making CADE-24 a success. I am very grateful to the members of the Program Committee and the external reviewers for carefully reviewing and evaluating the papers. On behalf of the Program Comittee, I thank Andrei Voronkov for the EasyChair system. I thank all authors who submitted papers, all participants of the conference, the invited speakers, the distinguished lecturers, the tutorial speakers, and the workshop organizers. CADE-24 would not have been possible without the dedicated work of the Organizing Committee. First and foremost, Neil Murray and Chris Lynch did a terrific job as Conference Chairs, planning and supervising all the conference events. Christoph Benzmüller and Peter Baumgartner were relentless as Workshop and Competition Chair, and Tutorial Chair, respectively. Heartfelt thanks go to Grant Olney Passmore for being such a proactive Publicity Chair, doing far more than the call of duty. I thank Geoff Sutcliffe for organizing CASC and also helping with publicity. Special thanks go to Catherine Zawadzki and the other personnel of the Crowne Plaza Hotel in Lake Placid, where all conference activities were held.

April 2013 Maria Paola Bonacina

Organization

Program Committee

Alessandro Armando	Università di Genova and FBK Trento, Italy
Peter Baumgartner	NICTA and Australian National University, Australia
Christoph Benzmüller	Freie Universität Berlin, Germany
Maria Paola Bonacina **(Chair)**	Università degli Studi di Verona, Italy
Cristina Borralleras	Universitat de Vic, Spain
Thierry Boy De La Tour	Université de Grenoble, France
Évelyne Contejean	CNRS and Université de Paris-Sud, France
Leonardo De Moura	Microsoft Research, USA
Stéphanie Delaune	École Normale Supérieure de Cachan, France
Clare Dixon	University of Liverpool, UK
Pascal Fontaine	Université de Lorraine and LORIA, France
Ulrich Furbach	Universität Koblenz-Landau, Germany
Ruben Gamboa	University of Wyoming, USA
Jürgen Giesl	RWTH Aachen, Germany
Paul B. Jackson	University of Edinburgh, UK
Predrag Janičić	Univerzitet u Beogradu, Serbia
Hélène Kirchner	INRIA Rocquencourt, France
Konstantin Korovin	University of Manchester, UK
K. Rustan M. Leino	Microsoft Research, USA
Christopher A. Lynch	Clarkson University, USA
César A. Muñoz	NASA Langley, USA
Neil V. Murray	University at Albany - SUNY, USA
Lawrence C. Paulson	University of Cambridge, UK
Frank Pfenning	Carnegie Mellon University, USA
Brigitte Pientka	McGill University, Canada
David A. Plaisted	University of North Carolina at Chapel Hill, USA
Christophe Ringeissen	LORIA and INRIA Nancy-Grand Est, France
Ulrike Sattler	University of Manchester, UK
Renate A. Schmidt	University of Manchester, UK
Manfred Schmidt-Schauß	J.W. Goethe Universität, Germany
Stephan Schulz	Technische Universität München, Germany
Viorica Sofronie-Stokkermans	Universität Koblenz-Landau and Max Planck Institut für Informatik, Germany
Ashish Tiwari	SRI International, USA
Uwe Waldmann	Max Planck Institut für Informatik, Germany

Christoph Weidenbach	Max Planck Institut für Informatik, Germany
Jian Zhang	Chinese Academy of Sciences, P.R. China

Organizing Committee

Conference Chairs

Neil V. Murray	University at Albany – SUNY, USA and
Christopher A. Lynch	Clarkson University, USA

Program Chair

Maria Paola Bonacina	Università degli Studi di Verona, Italy

Workshop and Competition Chair

Christoph Benzmüller	Freie Universität Berlin, Germany

Tutorial Chair

Peter Baumgartner	NICTA and Australian National University, Australia

Publicity Chair

Grant Olney Passmore	Cambridge University and Edinburgh University, UK

Additional Reviewers

Vincent Aravantinos
Mauricio Ayala-Rincón
Hicham Bensaid
Armin Biere
Nikolaj Bjørner
Bruno Blanchet
James Brotherston
Florian Bruse
Damien Doligez
Dan Dougherty
Vijay D'Silva
Mnacho Echenim
Vijay Ganesh
Rajeev Goré
Pieter Hooimeijer
Matthias Horbach
Ullrich Hustadt
Swen Jacobs
Yuncheng Jiang
Moa Johansson
Mark Kaminski
Mohammad Khodadadi
Patrick Koopmann
Pascal Lafourcade
Manuel Lamotte-Schubert
Daniel Le Berre
Etienne Lozes
Feifei Ma
Filip Marić
Andrew Matusiewicz
Charles Morisset
Anthony Narkawicz
Juan Antonio Navarro Pérez
Mladen Nikolić
Peter O'Hearn
Jens Otten

Fabio Papacchini
Grant Olney Passmore
Dirk Pattinson
Nicolas Peltier
Björn Pelzer
Florian Rabe
Silvio Ranise
Conrad Rau
Erik Rosenthal
Kristin Yvonne Rozier
Albert Rubio
Andrey Rybalchenko
David Sabel
Thomas Schneider
Mark E. Stickel
Adam Strzebonski
Thomas Sturm
Geoff Sutcliffe
Andreas Teucke
René Thiemann
Dmitry Tishkovsky
Dmitry Tsarkov
Heng Zhang
Wenhui Zhang

Board of Trustees of CADE Inc.

Franz Baader (President)	Technische Universität Dresden, Germany
Maria Paola Bonacina (PC Chair)	Università degli Studi di Verona, Italy
Amy Felty	University of Ottawa, Canada
Martin Giese (Secretary)	Universitetet i Oslo, Norway
Bernhard Gramlich	Technische Universität Wien, Austria
Neil V. Murray (Treasurer)	University at Albany - SUNY, USA
Lawrence C. Paulson	University of Cambridge, UK
Brigitte Pientka	McGill University, Canada
Renate A. Schmidt (Vice-President)	University of Manchester, UK
Viorica Sofronie-Stokkermans	Universität Koblenz-Landau and MPI für Informatik, Germany
Geoff Sutcliffe	University of Miami, USA
Christoph Weidenbach	MPI für Informatik, Germany

Board of the Association for Automated Reasoning

Martin Giese (Secretary)	Universitetet i Oslo, Norway
Neil V. Murray (CADE)	University at Albany - SUNY, USA
Hans Jürgen Ohlbach	Ludwig-Maximilians-Universität München, Germany
Brigitte Pientka (CADE)	McGill University, Canada
Larry Wos (President)	Argonne National Laboratory, USA

Sponsors

The CADE conference series is sponsored by CADE Inc., sub-corporation of the Association for Automated Reasoning. In addition, CADE-24 gratefully acknowledges support from the *Artificial Intelligence Journal* and Microsoft Research.

Table of Contents

One Logic to Use Them All

Jean-Christophe Filliâtre[1,2,3]

[1] CNRS
[2] LRI, Univ. Paris-Sud, Orsay, F-91405
[3] INRIA Saclay – Île-de-France, Orsay, F-91893

Abstract. Deductive program verification is making fast progress these days. One of the reasons is a tremendous improvement of theorem provers in the last two decades. This includes various kinds of automated theorem provers, such as ATP systems and SMT solvers, and interactive proof assistants. Yet most tools for program verification are built around a single theorem prover. Instead, we defend the idea that a collaborative use of several provers is a key to easier and faster verification.

This paper introduces a logic that is designed to target a wide set of theorem provers. It is an extension of first-order logic with polymorphism, algebraic data types, recursive definitions, and inductive predicates. It is implemented in the tool Why3, and has been successfully used in the verification of many non-trivial programs.

1 Introduction

The idea behind deductive program verification [20] is to break down the correctness of a program to a set of logical formulas and to prove them valid. A tremendous improvement of theorem provers in the last two decades now allows this idea to scale up. Projects such as CompCert [31] and seL4 [26] show how interactive proof assistants (resp. Coq and Isabelle) can be successfully used to tackle large program verifications. Even more impressive is the progress in automated provers[1], notably the so-called *SMT revolution*. Designed with program verification in mind, SMT solvers have led to powerful verification tools, such as VCC [39], Frama-C [23], Dafny [29], or VeriFast [25], to mention only a few. Yet we note that most of these tools are built on top of a single automated prover (*e.g.* Z3 in the case of VCC, Dafny, or VeriFast) or a single dedicated prover to handle a specific program logic (*e.g.* B method [1] or KIV [37]).

We rather defend the idea that program verification should be done using as many theorem provers as possible, including those that were not necessarily designed with program verification in mind (*e.g.* ATP systems). We are developing the tool Why3 [10,22] to implement this idea. Both the specification logic of Why3 and its programming language, WhyML, are used as intermediate languages in tools such as Frama-C [23], Krakatoa [33], Easycrypt [4], and GNATprove [14]. WhyML is also used directly to implement data structures and algorithms that can later be translated to executable OCaml code. Our gallery of

[1] We use *automated provers* to denote both SMT solvers and ATP systems.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 1–20, 2013.

verified programs (`http://toccata.lri.fr/gallery/`) currently contains more than 80 entries. These examples show the benefits of our approach. Indeed, it is often the case that several theorem provers are successfully used in proving all goals, but none can prove all of them by itself. In particular, an interactive theorem prover can be used to discharge a complex lemma (for instance one requiring induction), while the remaining goals can all be discharged automatically.

Designing a logic to target a wide set of theorem provers is not that easy. We have come up with a *logic of compromise*, that is not as rich as the logic of proof assistants such as Coq or PVS, yet richer than the usual logic of automated theorem provers. Our logic is an extension of first-order logic with rank-1 polymorphism, algebraic data types, recursive definitions, and inductive predicates. The purpose of this paper is to define this logic (Section 2), and to explain the processes by which Why3 translates it to the input format of fifteen theorem provers (Section 3). Finally, Section 4 presents experimental results obtained in the context of program verification. We conclude with related work and perspectives.

2 A Polymorphic First-Order Logic

This section describes the logic of Why3 as faithfully as possible. Earlier work [11] only describes the extension of first-order logic with polymorphism. Here we also consider algebraic data types, recursive definitions, and inductive predicates.

2.1 Syntax

A type symbol is simply a name $\mathtt{t}$ and a type arity $n \in \mathbb{N}$. We write it $\mathtt{t}\langle\alpha_1, \ldots, \alpha_n\rangle$ where the α_i are type variables. Names $\alpha_1, \ldots, \alpha_n$ are not relevant but we adopt an homogeneous syntax for all symbols. We note d_t such a declaration:

$$d_t \quad ::= \quad \mathtt{t}\langle\alpha, \ldots, \alpha\rangle \quad \text{type symbol declaration}$$

When $n = 0$, we say that $\mathtt{t}$ is a *monomorphic type symbol* and we simply write $\mathtt{t}$ instead of $\mathtt{t}\langle\rangle$. In the following, let Σ_T be a set of type symbols. We assume that Σ_T contains at least the two monomorphic type symbols int and real. Types are built from type variables and type symbols:

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{type variable} \\ & | & \mathtt{t}\langle\tau, \ldots, \tau\rangle & \text{type symbol application} \end{array}$$

We note $tv(\tau)$ the set of type variables of type τ. When $tv(\tau) = \emptyset$, we say that τ is a *sort*.

Function and predicate symbols are declared, possibly with polymorphic types, as follows:

$$\begin{array}{rcll} d_f & ::= & \mathsf{f}\langle\alpha, \ldots, \alpha\rangle(\tau, \ldots, \tau) : \tau & \text{function symbol declaration} \\ d_p & ::= & \mathsf{p}\langle\alpha, \ldots, \alpha\rangle(\tau, \ldots, \tau) & \text{predicate symbol declaration} \end{array}$$

In the following, Σ_F denotes a set of function symbols and Σ_P a set of predicate symbols. We assume that Σ_P contains at least a polymorphic predicate $=\langle\alpha\rangle(\alpha,\alpha)$ that denotes equality. Terms and formulas are then built from function and predicate symbols according to the syntax given in Fig. 1–3. Note that syntax for terms and formulas are mutually recursive, since a conditional term expression `if` f `then` t_1 `else` t_2 involves a formula f.

We note $fv(t)$ (resp. $fv(f)$, $fv(p)$) the set of free variables of a term t (resp. a formula f, a pattern p). Definitions of $fv(t)$ and $fv(f)$ are standard, variables being bound by `let`, quantifiers, and patterns. A pattern p binds all variables in $fv(p)$, which is defined as follows:

$$\begin{aligned} fv(x_\tau) &= \{x_\tau\} \\ fv(\mathsf{f}\langle\tau_1,\ldots,\tau_m\rangle(p_1,\ldots,p_n)) &= fv(p_1)\cup\cdots\cup fv(p_n) \\ fv(\_) &= \emptyset \\ fv(p_1 \mid p_2) &= fv(p_1)\cup fv(p_2) \\ fv(p \texttt{ as } x_\tau) &= fv(p)\cup\{x_\tau\} \end{aligned}$$

The type checking rule for pattern $p_1 \mid p_2$, given in next section, imposes that $fv(p_1)$ and $fv(p_2)$ are equal. Thus the definition above actually simplifies to $fv(p_1 \mid p_2) = fv(p_1) = fv(p_2)$.

A *signature* Σ denotes a triple $(\Sigma_T, \Sigma_F, \Sigma_P)$. A *context* Γ extends a signature Σ with definitions for some of its symbols. A definition d introduces either algebraic data types, recursive definitions, or inductive predicates, as follows:

d	::=	`datatype` a `with` ... `with` a	algebraic data types
	|	`recursive` δ `with` ... `with` δ	recursive definitions
	|	`inductive` i `with` ... `with` i	inductive predicates
a	::=	$d_t = d_f \mid \ldots \mid d_f$	algebraic data type
δ	::=	`function` $\mathsf{f}\langle\alpha,\ldots,\alpha\rangle(x_\tau,\ldots,x_\tau) : \tau = t$	function definition
	|	`predicate` $\mathsf{p}\langle\alpha,\ldots,\alpha\rangle(x_\tau,\ldots,x_\tau) = f$	predicate definition
i	::=	$d_p = f \mid \ldots \mid f$	inductive predicate

Example. Let Σ_T be the following set of type symbols:

$$\Sigma_T = \{\mathsf{int},\ \mathsf{real},\ \mathsf{nat},\ \mathsf{list}\langle\alpha\rangle,\ \mathsf{pair}\langle\alpha_1,\alpha_2\rangle\}.$$

Let Σ_F and Σ_P be the following sets of function and predicate symbols:

$$\begin{aligned} \Sigma_F = \{&\mathsf{O} : \mathsf{nat},\ \mathsf{S}(\mathsf{nat}) : \mathsf{nat}, \\ &\mathsf{Nil}\langle\alpha\rangle : \mathsf{list}\langle\alpha\rangle,\ \mathsf{Cons}\langle\alpha\rangle(\alpha,\mathsf{list}\langle\alpha\rangle) : \mathsf{list}\langle\alpha\rangle,\ \mathsf{length}\langle\alpha\rangle(\mathsf{list}\langle\alpha\rangle) : \mathtt{nat}, \\ &\mathsf{Pair}\langle\alpha_1,\alpha_2\rangle(\alpha_1,\alpha_2) : \mathsf{pair}\langle\alpha_1,\alpha_2\rangle\ \} \\ \Sigma_P = \{&=\langle\alpha\rangle(\alpha,\alpha),\ \mathsf{even}(\mathsf{nat})\} \end{aligned}$$

t	::=	c_{int}	literal integer constant
	|	c_{real}	literal real constant
	|	x_τ	variable
	|	$\mathsf{f}\langle\tau,\ldots,\tau\rangle(t,\ldots,t)$	function symbol application
	|	$\texttt{let}\ x_\tau = t\ \texttt{in}\ t$	local binding
	|	$\texttt{if}\ f\ \texttt{then}\ t\ \texttt{else}\ t$	conditional expression
	|	$\texttt{match}\ t\ \texttt{with}\ p \to t\ \texttt{\|}\ \ldots\ \texttt{\|}\ p \to t\ \texttt{end}$	pattern matching

Fig. 1. Syntax for terms

f	::=	$\mathsf{p}\langle\tau,\ldots,\tau\rangle(t,\ldots,t)$	predicate symbol application
	|	$\forall x_\tau.\, f$	universal quantification
	|	$\exists x_\tau.\, f$	existential quantification
	|	$f \wedge f$	conjunction
	|	$f \vee f$	disjunction
	|	$f \Rightarrow f$	implication
	|	$f \Leftrightarrow f$	equivalence
	|	$\texttt{not}\ f$	negation
	|	$\texttt{true}$	tautology
	|	$\texttt{false}$	absurdity
	|	$\texttt{let}\ x_\tau = t\ \texttt{in}\ f$	local binding
	|	$\texttt{if}\ f\ \texttt{then}\ f\ \texttt{else}\ f$	conditional expression
	|	$\texttt{match}\ t\ \texttt{with}\ p \to f\ \texttt{\|}\ \ldots\ \texttt{\|}\ p \to f\ \texttt{end}$	pattern matching

Fig. 2. Syntax for formulas

p	::=	x_τ	variable
	|	$\mathsf{f}\langle\tau,\ldots,\tau\rangle(p,\ldots,p)$	constructor application
	|	$\_$	catch all
	|	$p\ \texttt{\|}\ p$	or pattern
	|	$p\ \texttt{as}\ x_\tau$	binding

Fig. 3. Syntax for patterns

Let Γ be the extension of the signature above with the following set of definitions:

$$
\begin{array}{l}
\texttt{datatype}\ \mathsf{nat} = \mathsf{O} \mid \mathsf{S}(\mathsf{nat}) \\
\texttt{datatype}\ \mathsf{list}\langle\alpha\rangle = \mathsf{Nil}\langle\alpha\rangle \mid \mathsf{Cons}\langle\alpha\rangle(\alpha, \mathsf{list}\langle\alpha\rangle) \\
\texttt{recursive function}\ \mathsf{length}\langle\alpha\rangle(l_{\mathsf{list}\langle\alpha\rangle}) : \mathsf{nat} = \\
\quad \texttt{match}\ l_{\mathsf{list}\langle\alpha\rangle}\ \texttt{with}\ \mathsf{Nil}\langle\alpha\rangle \to \mathsf{O} \mid \mathsf{Cons}\langle\alpha\rangle(\_, r_{\mathsf{list}\langle\alpha\rangle}) \to \mathsf{S}(\mathsf{length}(r_{\mathsf{list}\langle\alpha\rangle}))\ \texttt{end} \\
\texttt{inductive}\ \mathsf{even}(\mathsf{nat}) = \\
\quad \mathsf{even}(\mathsf{O}) \mid \forall n_{\mathsf{nat}}.\, \mathsf{even}(n_{\mathsf{nat}}) \Rightarrow \mathsf{even}(\mathsf{S}(\mathsf{S}(n_{\mathsf{nat}})))
\end{array}
$$

Here and below, we omit type annotations when they are obvious from the context.

2.2 Type Checking

For a signature Σ to be well-formed, any type expression τ occurring in a declaration of Σ_F or Σ_P must be well-typed, that is $\Sigma \vdash \tau$. Additionally, the free type variables of τ must be included in the type variables $\langle\alpha_1, \ldots, \alpha_m\rangle$ of the declaration. In the following, we only consider signatures that are well-formed. Let Γ be a context extending a signature Σ with definitions. For Γ to be well-formed, any symbol from a definition must appear in Σ, and no symbol can be defined more that once.

Type checking of types, terms, formulas, and patterns is straightforward. Judgements are $\Sigma \vdash \tau$ (type τ is well-formed in Σ), $\Sigma \vdash t : \tau$ (term t is well-formed and of type τ), $\Sigma \vdash f$ (formula f is well-formed), and $\Sigma \vdash p : \tau$ (pattern p is well-formed and of type τ), respectively. Typing rules are given in Fig. 4–7. Note that, contrary to Hindley-Milner type system, the `let` binder does not generalize the type of the term that is bound. Thus polymorphism is introduced by symbols, but not by local definitions. In addition to these rules, we also check that, for any pattern matching expression $\texttt{match}\ t\ \texttt{with}\ p_1 \to . \mid \ldots \mid p_n \to .\ \texttt{end}$ (in terms or formulas), t has type $\mathsf{t}\langle\tau_1, \ldots, \tau_m\rangle$ where type t is defined as an algebraic data type and patterns $p_1, \ldots, p_n$ cover any possible value for term t.

Finally, we perform the following checks on definitions.

Algebraic Data Types. Within a block of mutually-defined algebraic data types $\texttt{datatype}\ a_1\ \texttt{with}\ \ldots\ \texttt{with}\ a_k$, each algebraic data type definition a_i must have the form

$$
\begin{array}{rl}
\mathsf{t}_i\langle\alpha_1, \ldots, \alpha_m\rangle & = \mathsf{f}_1\langle\alpha_1, \ldots, \alpha_m\rangle(\ldots) : \mathsf{t}_i\langle\alpha_1, \ldots, \alpha_m\rangle \\
& \mid \ \ldots \\
& \mid \mathsf{f}_l\langle\alpha_1, \ldots, \alpha_m\rangle(\ldots) : \mathsf{t}_i\langle\alpha_1, \ldots, \alpha_m\rangle
\end{array}
$$

that is, all constructors f_j share the same type parameters as type t_i and all have return type $\mathsf{t}_i\langle\alpha_1, \ldots, \alpha_m\rangle$. Additionally, we require that all types t_i are *inhabited* according to the following definition: Type t_i is inhabited if it has at least one constructor $\mathsf{f}_j\langle\alpha_1, \ldots, \alpha_m\rangle(\tau_1, \ldots, \tau_{n_j})$ such that all types τ_l are inhabited. Since types τ_l may recursively involve types from data type definitions, such a definition is to be understood as a least fixed point. Types from Σ that are

$$\frac{\mathsf{t}\langle\alpha_1,\ldots,\alpha_m\rangle \in \Sigma \quad \Sigma \vdash \tau_i}{\Sigma \vdash \mathsf{t}\langle\tau_1,\ldots,\tau_m\rangle}$$

Fig. 4. Typing rule for types

$$\frac{}{\Sigma \vdash c_{\mathsf{int}} : \mathsf{int}} \qquad \frac{}{\Sigma \vdash c_{\mathsf{real}} : \mathsf{real}} \qquad \frac{\Sigma \vdash \tau}{\Sigma \vdash x_\tau : \tau}$$

$$\frac{\begin{array}{c}\mathsf{f}\langle\alpha_1,\ldots,\alpha_m\rangle(\tau'_1,\ldots,\tau'_n) : \tau \in \Sigma \quad \Sigma \vdash \tau_i \\ \sigma = \{\alpha_1 \mapsto \tau_1,\ldots,\alpha_m \mapsto \tau_m\} \quad \Sigma \vdash t_i : \sigma(\tau'_i)\end{array}}{\Sigma \vdash \mathsf{f}\langle\tau_1,\ldots,\tau_m\rangle(t_1,\ldots,t_n) : \sigma(\tau)}$$

$$\frac{\Sigma \vdash t_1 : \tau \quad \Sigma \vdash t_2 : \tau_2}{\Sigma \vdash \mathtt{let}\ x_\tau = t_1\ \mathtt{in}\ t_2 : \tau_2} \qquad \frac{\Sigma \vdash f \quad \Sigma \vdash t_1 : \tau \quad \Sigma \vdash t_2 : \tau}{\Sigma \vdash \mathtt{if}\ f\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 : \tau}$$

$$\frac{\Sigma \vdash t : \mathsf{t}\langle\tau_1,\ldots,\tau_m\rangle \quad \Sigma \vdash p_i : \mathsf{t}\langle\tau_1,\ldots,\tau_m\rangle \quad \Sigma \vdash t_i : \tau}{\Sigma \vdash \mathtt{match}\ t\ \mathtt{with}\ p_1 \to t_1\ \mid\ \ldots\ \mid\ p_n \to t_n\ \mathtt{end} : \tau}$$

Fig. 5. Typing rules for terms

$$\frac{\begin{array}{c}\mathsf{p}\langle\alpha_1,\ldots,\alpha_m\rangle(\tau'_1,\ldots,\tau'_n) \in \Sigma \quad \Sigma \vdash \tau_i \\ \sigma = \{\alpha_1 \mapsto \tau_1,\ldots,\alpha_m \mapsto \tau_m\} \quad \Sigma \vdash t_i : \sigma(\tau'_i)\end{array}}{\Sigma \vdash \mathsf{p}\langle\tau_1,\ldots,\tau_m\rangle(t_1,\ldots,t_n)}$$

$$\frac{}{\Sigma \vdash \mathtt{true}} \qquad \frac{}{\Sigma \vdash \mathtt{false}} \qquad \frac{\Sigma \vdash f_1 \quad \Sigma \vdash f_2 \quad \circ \in \{\wedge,\vee,\Rightarrow,\Leftrightarrow\}}{\Sigma \vdash f_1 \circ f_2}$$

$$\frac{\Sigma \vdash \tau \quad \Sigma \vdash f}{\Sigma \vdash \forall x_\tau.\, f} \qquad \frac{\Sigma \vdash \tau \quad \Sigma \vdash f}{\Sigma \vdash \exists x_\tau.\, f} \qquad \frac{\Sigma \vdash f}{\Sigma \vdash \mathtt{not}\ f}$$

$$\frac{\Sigma \vdash t : \tau \quad \Sigma \vdash f}{\Sigma \vdash \mathtt{let}\ x_\tau = t\ \mathtt{in}\ f} \qquad \frac{\Sigma \vdash f_1 \quad \Sigma \vdash f_2 \quad \Sigma \vdash f_3}{\Sigma \vdash \mathtt{if}\ f_1\ \mathtt{then}\ f_2\ \mathtt{else}\ f_3}$$

$$\frac{\Sigma \vdash t : \mathsf{t}\langle\tau_1,\ldots,\tau_m\rangle \quad \Sigma \vdash p_i : \mathsf{t}\langle\tau_1,\ldots,\tau_m\rangle \quad \Sigma \vdash f_i}{\Sigma \vdash \mathtt{match}\ t\ \mathtt{with}\ p_1 \to f_1\ \mid\ \ldots\ \mid\ p_n \to f_n\ \mathtt{end}}$$

Fig. 6. Typing rules for formulas

$$\frac{\Sigma \vdash \tau}{\Sigma \vdash x_\tau : \tau} \qquad \frac{\Sigma \vdash \tau}{\Sigma \vdash \_ : \tau}$$

$$\frac{\begin{array}{c}f\langle\alpha_1,\ldots,\alpha_m\rangle(\tau'_1,\ldots,\tau'_n) : \tau \in \Sigma \quad \Sigma \vdash \tau_i \quad \sigma = \{\alpha_1 \mapsto \tau_1,\ldots,\alpha_m \mapsto \tau_m\} \\ i \neq j \Rightarrow fv(p_i) \cap fv(p_j) = \emptyset \quad \Sigma \vdash p_i : \sigma(\tau'_i)\end{array}}{\Sigma \vdash f\langle\tau_1,\ldots,\tau_m\rangle(p_1,\ldots,p_n) : \sigma(\tau)}$$

$$\frac{fv(p_1) = fv(p_2) \quad \Sigma \vdash p_i : \tau}{\Sigma \vdash p_1 \mid p_2 : \tau} \qquad \frac{x_\tau \notin fv(p) \quad \Sigma \vdash p : \tau}{\Sigma \vdash p\ \mathtt{as}\ x_\tau : \tau}$$

Fig. 7. Typing rules for patterns

not part of data type definitions are assumed to be inhabited. As a consequence of this definition, an algebraic data type must have at least one constructor.

Recursive Definitions. A block of mutually recursive definitions $\texttt{recursive}\ \delta_1\ \texttt{with}\ \ldots\ \texttt{with}\ \delta_k$ is well-formed if each definition δ_i is well-typed. A function definition

$$\texttt{function}\ \mathsf{f}\langle\alpha,\ldots,\alpha\rangle(x_\tau,\ldots,x_\tau) : \tau = t$$

is well-typed if $\Sigma \vdash t : \tau$, and a predicate definition

$$\texttt{predicate}\ \mathsf{p}\langle\alpha,\ldots,\alpha\rangle(x_\tau,\ldots,x_\tau) = f$$

is well-typed if $\Sigma \vdash f$. Additionally, all free variables (including type variables) in t and f must belong to the symbol declaration. Finally, we require such definitions to *terminate*, according to the following criterion: there must exist a lexicographic order of arguments that guarantees a structural descent (over algebraic data types).

Inductive Predicates. An inductive definition

$$\mathsf{p}\langle\alpha_1,\ldots,\alpha_m\rangle(\tau_1,\ldots,\tau_n) = f_1 \mid \ldots \mid f_k$$

is well-formed if each clause f_i is a closed formula, well-typed *i.e.* $\Sigma \vdash f_i$, that belongs to the following grammar:

$$\begin{array}{rcl} f_0 & ::= & \mathsf{p}\langle\alpha_1,\ldots,\alpha_m\rangle(t_1,\ldots,t_n) \\ & \mid & f \Rightarrow f_0 \\ & \mid & \forall x_\tau.\, f_0 \\ & \mid & \texttt{let}\ x_\tau = t\ \texttt{in}\ f_0 \end{array}$$

Additionally, we require all inductive predicates from that block of inductive definitions to appear in strictly positive positions in the formulas on the left side of $\Rightarrow$, so as to ensure the existence of a least fixpoint.

In the following, we only consider contexts that are well-formed according to the typing rules we just introduced.

2.3 Semantics

We recall that a sort is a monomorphic type, that is, a type τ such that $tv(\tau) = \emptyset$. In the following, we use notation $\boldsymbol{\alpha}$ (resp. $\boldsymbol{s}$, $\boldsymbol{x}$, $\boldsymbol{t}$) for a vector of type variables (resp. sorts, variables, terms). Given a signature, a *pre-interpretation* $[\![.]\!]$ is defined as follows:

- Each sort s is interpreted as a non-empty domain $[\![s]\!]$. Sort int is interpreted as $\mathbb{Z}$ and sort real as $\mathbb{R}$.

- Given a function symbol $\mathsf{f}\langle\boldsymbol{\alpha}\rangle(\tau_1, \ldots, \tau_n) : \tau$ and sorts $\boldsymbol{s}$, we interpret the instance $\mathsf{f}\langle\boldsymbol{s}\rangle$ as a function $[\![\mathsf{f}\langle\boldsymbol{s}\rangle]\!]$ of type

$$[\![\sigma(\tau_1)]\!] \times \cdots \times [\![\sigma(\tau_n)]\!] \to [\![\sigma(\tau)]\!]$$

  where σ maps the type variables $\boldsymbol{\alpha}$ to the sorts $\boldsymbol{s}$.
- Given a predicate symbol $\mathsf{p}\langle\boldsymbol{\alpha}\rangle(\tau_1, \ldots, \tau_n)$ and sorts $\boldsymbol{s}$, we interpret the instance $\mathsf{p}\langle\boldsymbol{s}\rangle$ as a function $[\![\mathsf{p}\langle\boldsymbol{s}\rangle]\!]$ of type

$$[\![\sigma(\tau_1)]\!] \times \cdots \times [\![\sigma(\tau_n)]\!] \to \{\bot, \top\}$$

  where σ maps the type variables $\boldsymbol{\alpha}$ to the sorts $\boldsymbol{s}$. For each sort s, the predicate $=\langle s\rangle$ is interpreted as the equality over $[\![s]\!]$.
- For any algebraic data type $\mathsf{t}\langle\boldsymbol{\alpha}\rangle$, with constructors $\mathsf{f}_1\langle\boldsymbol{\alpha}\rangle, \ldots, \mathsf{f}_l\langle\boldsymbol{\alpha}\rangle$, we require $[\![\mathsf{t}\langle\boldsymbol{s}\rangle]\!]$ to be the free algebra generated by $[\![\mathsf{f}_1\langle\boldsymbol{s}\rangle]\!], \ldots, [\![\mathsf{f}_l\langle\boldsymbol{s}\rangle]\!]$, that is

$$\text{for } i \neq j, [\![\mathsf{f}_i\langle\boldsymbol{s}\rangle]\!](\boldsymbol{t}) \neq [\![\mathsf{f}_j\langle\boldsymbol{s}\rangle]\!](\boldsymbol{u}) \tag{1}$$

$$[\![\mathsf{f}_i\langle\boldsymbol{s}\rangle]\!](\boldsymbol{t}) = [\![\mathsf{f}_i\langle\boldsymbol{s}\rangle]\!](\boldsymbol{u}) \Rightarrow \boldsymbol{t} = \boldsymbol{u} \tag{2}$$

$$\forall x \in [\![\mathsf{t}\langle\boldsymbol{s}\rangle]\!],\ \exists i, \exists \boldsymbol{t},\ x = [\![\mathsf{f}_i\langle\boldsymbol{s}\rangle]\!](\boldsymbol{t}) \tag{3}$$

  In the following, isf_i denotes a predicate that identifies values in $[\![\mathsf{t}\langle\boldsymbol{s}\rangle]\!]$ that are applications of f_i, and $\mathsf{projf}_{i,j}$ returns the j-th argument of such an application.

Given a pre-interpretation, a *valuation* v maps each type variable α to a sort $v(\alpha)$, and each variable x_τ to some element of $[\![v(\tau)]\!]$. Given a pre-interpretation and a valuation v, a term t of type τ is interpreted as an element $[\![t]\!]_v \in [\![v(\tau)]\!]$ and a formula f is interpreted as a Boolean value $[\![f]\!]_v$, according to the definitions given in Fig. 8–9. Note that pattern matching is compiled away as show in Fig. 10. Operator M turns a `match` construct into elementary tests. More precisely, $M(t, p, b, h)$ filters value t against pattern p and returns b in case of success and h in case of failure. Since type checking ensures that any pattern matching is exhaustive, the *error* term cannot actually appear in the result of M. It is only used to simplify the definition and use of function M.

An *interpretation* is a pre-interpretation that is consistent with recursive and inductive definitions, that is:

- For any recursive definition $\texttt{function}\ \mathsf{f}\langle\boldsymbol{\alpha}\rangle(\boldsymbol{x}) : \tau = t$ and any $\boldsymbol{s}$, we require $[\![\mathsf{f}\langle\boldsymbol{s}\rangle]\!]$ to be such that, for all $\boldsymbol{t}$, $[\![\mathsf{f}\langle\boldsymbol{s}\rangle]\!](\boldsymbol{t}) = [\![t]\!]_v$ where v maps the $\boldsymbol{\alpha}$ to the $\boldsymbol{s}$ and the $\boldsymbol{x}$ to the $\boldsymbol{t}$ (and similarly for a predicate definition).
- For any inductive definition $\mathsf{p}\langle\boldsymbol{\alpha}\rangle(\boldsymbol{\tau}) = f_1 \mid \ldots \mid f_l$ and any $\boldsymbol{s}$, we require $[\![\mathsf{p}\langle\boldsymbol{s}\rangle]\!]$ to be the least predicate such that $[\![f_1]\!]_v, \ldots, [\![f_l]\!]_v$ hold where v maps the $\boldsymbol{\alpha}$ to the $\boldsymbol{s}$.

$$
\begin{aligned}
[\![n_{\mathsf{int}}]\!]_v &= n \\
[\![r_{\mathsf{real}}]\!]_v &= r \\
[\![x_\tau]\!]_v &= v(x_\tau) \\
[\![\mathsf{f}\langle\tau_1,\ldots,\tau_m\rangle(t_1,\ldots,t_n)]\!]_v &= [\![\mathsf{f}\langle v(\tau_1),\ldots,v(\tau_m)\rangle]\!]([\![t_1]\!]_v,\ldots,[\![t_n]\!]_v) \\
[\![\texttt{let } x_\tau = t_1 \texttt{ in } t_2]\!]_v &= [\![t_2]\!]_{v[x_\tau \mapsto [\![t_1]\!]_v]} \\
[\![\texttt{if } f \texttt{ then } t_1 \texttt{ else } t_2]\!]_v &= [\![t_1]\!]_v \text{ if } [\![f]\!]_v = \top, \\
&= [\![t_2]\!]_v \text{ otherwise} \\
[\![\texttt{match } t \texttt{ with } p_1 \to t_1 \texttt{ | } \ldots \texttt{ | } p_n \to t_n \texttt{ end}]\!]_v &= [\![\, M(t,p_1,t_1,M(t,p_2,t_2, \\
&\qquad \ldots,M(t,p_n,t_n,\mathit{error})))]\!]_v
\end{aligned}
$$

Fig. 8. Interpretation of terms

$$
\begin{aligned}
[\![\mathsf{p}\langle\tau_1,\ldots,\tau_m\rangle(t_1,\ldots,t_n)]\!]_v &= [\![\mathsf{p}\langle v(\tau_1),\ldots,v(\tau_m)\rangle]\!]([\![t_1]\!]_v,\ldots,[\![t_n]\!]_v) \\
[\![\forall x_\tau.\, f]\!]_v &= \\
[\![\exists x_\tau.\, f]\!]_v &= \\
[\![f_1 \circ f_2]\!]_v &= [\![f_1]\!]_v \circ [\![f_2]\!]_v \text{ where } \circ \in \{\wedge,\vee,\Rightarrow,\Leftrightarrow\} \\
[\![\texttt{not } f]\!]_v &= \neg[\![f]\!]_v \\
[\![\texttt{true}]\!]_v &= \top \\
[\![\texttt{false}]\!]_v &= \bot \\
[\![\texttt{let } x_\tau = t_1 \texttt{ in } f_2]\!]_v &= [\![f_2]\!]_{v[x_\tau \mapsto [\![t_1]\!]_v]} \\
[\![\texttt{if } f_1 \texttt{ then } f_2 \texttt{ else } f_3]\!]_v &= [\![f_2]\!]_v \text{ if } [\![f_1]\!]_v = \top, \\
&= [\![f_3]\!]_v \text{ otherwise} \\
[\![\texttt{match } t \texttt{ with } p_1 \to f_1 \texttt{ | } \ldots \texttt{ | } p_n \to f_n \texttt{ end}]\!]_v &= [\![\, M(t,p_1,f_1,M(t,p_2,f_2, \\
&\qquad \ldots,M(t,p_n,f_n,\mathit{error})))]\!]_v
\end{aligned}
$$

Fig. 9. Interpretation of formulas

$$
\begin{aligned}
M(t,x_\tau,b,h) &= \texttt{let } x_\tau = t \texttt{ in } b \\
M(t,\mathsf{f}_i(),b,h) &= \texttt{if } \mathsf{isf}_i(t) \texttt{ then } b \texttt{ else } h \\
M(t,\mathsf{f}_i(p_1,\ldots,p_n),b,h) &= \texttt{if } \mathsf{isf}_i(t) \texttt{ then } M(\mathsf{projf}_{i,1}(t),p_1,M(\mathsf{projf}_{i,2}(t),p_2,\ldots,h),h) \texttt{ else } h \\
M(t,\_,b,h) &= b \\
M(t,p_1 \texttt{ | } p_2,b,h) &= M(t,p_1,b,M(t,p_2,b,h)) \\
M(t,p \texttt{ as } x_\tau,b,h) &= \texttt{let } x_\tau = t \texttt{ in } M(t,p,b,h)
\end{aligned}
$$

Fig. 10. Interpretation of patterns

A set of closed formulas Δ is *satisfied* by $[\![.]\!]$ if and only if $[\![f]\!]_v = \top$ for every $f \in \Delta$ and every valuation v. (Note that only the mapping of type variables in v is relevant here, since formulas are closed.) A set of closed formulas Δ is *satisfiable* if and only if it is satisfied for some interpretation. Given a set of polymorphic closed axioms Δ and a closed formula f to be proved, we may assume that f is monomorphic (otherwise we simply replace type variables in f by fresh type constants). Then we say that formula f is a *logical consequence* of Δ if and only if the set $\Delta, \texttt{not}\ f$ is unsatisfiable.

Discussion. It is worth pointing out that our notion of interpretation does not prevent two distinct instances of a polymorphic symbol, say two types $\mathsf{t}\langle s_1\rangle$ and $\mathsf{t}\langle s_2\rangle$ or two functions $\mathsf{f}\langle s_1\rangle$ and $\mathsf{f}\langle s_2\rangle$, to be interpreted in two completely different ways. Even for an algebraic data type, say $\texttt{list}\langle\alpha\rangle$, we do not require $\mathsf{Nil}\langle s_1\rangle$ and $\mathsf{Nil}\langle s_2\rangle$ to be identical. We simply require $\texttt{list}\langle s_1\rangle$ to be the free algebra generated by $\mathsf{Nil}\langle s_1\rangle$ and $\mathsf{Cons}\langle s_1\rangle$, and similarly $\texttt{list}\langle s_2\rangle$ to be the free algebra generated by $\mathsf{Nil}\langle s_2\rangle$ and $\mathsf{Cons}\langle s_2\rangle$.

As a consequence, there is nothing wrong with an axiom defining a property of $\mathsf{f}\langle\mathsf{int}\rangle$, for some polymorphic function $\mathsf{f}\langle\alpha\rangle$, while all other instances are left uninterpreted. Even further, we could have two completely unrelated axioms for $\mathsf{f}\langle\mathsf{int}\rangle$ and $\mathsf{f}\langle\mathsf{real}\rangle$, *e.g.* that $\mathsf{f}\langle\mathsf{int}\rangle$ is the identity over type int while $\mathsf{f}\langle\mathsf{real}\rangle$ is the square root function.

Simply speaking, everything works as if we were using many-sorted logic with a possibly infinite set of simple sorts (int, $\mathsf{list}\langle\mathsf{int}\rangle$, $\mathsf{list}\langle\mathsf{real}\rangle$, $\mathsf{list}\langle\mathsf{list}\langle\mathsf{int}\rangle\rangle$, etc.), and a possibly infinite set of functions and predicates with simple types ($\mathsf{Nil}\langle\mathsf{int}\rangle$, $\mathsf{Nil}\langle\mathsf{real}\rangle$, $\mathsf{Cons}\langle\mathsf{list}\langle\mathsf{int}\rangle\rangle$, etc.), with suitable extensions for algebraic data types, recursive definitions, and inductive predicates.

2.4 Implementation

The logic we just described is implemented in Why3 at two different levels: an OCaml API and a surface language.

The OCaml API allows the user to build terms, patterns, formulas, declarations, and goals. The API is defensive: only well-typed values can be built, according to the typing rules of Sec. 2.2. Following the presentation above, one first builds symbols and then, later, possible definitions for these symbols. This way, terms, patterns, and formulas can be built as soon as symbols are available, without the need of passing around a context containing definitions.

On top of this API, Why3 provides a surface language. Fig. 11 contains an example of input file. Contrary to the API, when a definition is provided, we do not separate the signature and the definition. For instance, we simply write

```
type list 'a = Nil | Cons 'a (list 'a)
```

to simultaneously introduce the type symbol `list 'a`, the two function symbols `Nil` and `Cons` (its constructors), and the algebraic data type definition.

```
theory Example

  type nat = O | S nat

  type list 'a = Nil | Cons 'a (list 'a)

  function length (l: list 'a) : nat =
    match l with
    | Nil → O
    | Cons _ r → S (length r)
    end

  inductive even (nat) =
    | EvenO: even O
    | EvenS: forall m: nat. even m → even (S (S m))

  goal G: even (length (Cons O (Cons O Nil)))

end
```

Fig. 11. Example of Why3 input file

In the surface language, there are no angle brackets $\langle . \rangle$ anymore. First, type variables are not explicitly bound in symbol declarations. They are rather gathered, as the set of all type variables appearing in the symbol's type. For instance, the declaration

```
function length (l: list 'a) : nat = ...
```

introduces a function symbol `length` with one type variable. Second, type variables are not explicitly instantiated when a symbol is used. Instantiation is inferred from the arguments whenever possible. For instance, one simply writes

```
length (Cons 1 (Cons 2 Nil))
```

without having to pass $\langle \mathsf{int} \rangle$ to `Nil`, `Cons`, and `length`. When the instantiation cannot be computed from the arguments, it must be provided. A typical example is the following:

```
lemma L: length (Nil: list 'a) = O
```

Without the type cast, this would result in an `undefined type variable` error. In both the API and the surface language, there is no type inference, only type checking. In particular, quantified variables are given types (such as `forall m: nat` in Fig. 11). This is a deliberate choice: we could implement type inference, but we think that specifications are easier to read when types are given.

Among other features of the surface language of Why3 are type aliases (such as `type t = list int`) and tuple types. Type aliases are inlined systematically.

Tuple types are particular cases of algebraic data types, and are generated on the fly. There is also a syntax for record types (algebraic data types with a single constructor and named projection functions). Finally, logic declarations are organized in units called *theories*, as shown in Fig. 11. Theories can refer to and instantiate other theories. We do not discuss theories here; see [10] for details.

3 Targeting Multiple Provers

We now explain how Why3 translates the logic we just described into the native input format of external theorem provers. Currently, Why3 supports the following theorem provers:

- Proof assistants: Coq 8.4 [41] and PVS 6.0 [36];
- SMT solvers: Alt-Ergo 0.95.1 [7], CVC3 2.4.1 [3], CVC4 1.0 [2], Simplify 1.5.4 [19], Yices 1.0.38 [18] and Yices2 2.0.4, Z3 4.3.1 [17];
- ATP systems: E 1.6 [40], iProver 0.8.1 [27], SPASS 3.7 [42], Vampire 0.6 [38], Zenon 0.7.1 [13];
- Dedicated provers: Gappa 0.15.1 [16], Mathematica 8.0.

We only list the most recent versions that are supported; some older versions may be supported as well.

3.1 Tasks and Transformations

A central notion in Why3 is that of *task*. A task is a context Γ (symbols, possibly with definitions), a set of axioms Δ, and a formula to be proved. Tasks are massaged using *transformations* until they reach a subset of Why3's logic that coincides with the input format of a theorem prover. A transformation turns a task into a set of new tasks. Key transformations are the following:

- elimination of recursion, inductive predicates, algebraic data types and pattern matching, `if-then-else` construct, `let` binding;
- encoding of types, to target many-sorted or untyped logic [15,11].

In the process of proving a task, other transformations may be used, such as inlining, splitting, various kinds of simplification, or even induction (following Leino [30]). Why3 can be extended with new transformations via OCaml plugins. The OCaml API allows the user to build tasks and to apply transformations efficiently (thanks to memoization).

3.2 Drivers

The way a particular prover is handled in Why3 is controlled by a text file called a *driver*. Such a file lists the transformations that must be applied to a task prior to its transmission to the prover, defines the pretty-printer that must be

used when we are done with transformations, lists symbols and axioms that are built-in in the prover, and provides regular expressions to interpret the output of calls to the prover.

For instance, the driver for SPASS refers to the printer registered under the name `tptp-fof`, includes transformations such as `eliminate_algebraic` (to get rid of algebraic data types and pattern matching) and `encoding_tptp` (to encode types), and states that equality is built-in with syntax `(x=y)`, among other things.

Users may define new drivers, to add support for a new prover or to experiment with alternative ways of using a prover (*e.g.* not making use of a built-in theory, or using an alternate input format). For that purpose, Why3 can be extended with new pretty-printers via OCaml plug-ins. The OCaml API provides ways to load a driver and then to use it to call the corresponding theorem prover on a task. Drivers are described in more details in our earlier work [10].

Prover Specificities. There are several places where we have to be careful to avoid introducing inconsistencies in the process of translating from Why3 to theorem provers.

An example is integer division. Why3 standard library contains two theories: one for Euclidean division, and another where division rounds towards zero as in most programming languages. When it comes to using a built-in notion of division provided by some prover, we have to identify which one it is, if any, or to provide workarounds otherwise, if possible. For instance, Z3 provides Euclidean division and modulo, but CVC4 provides a division that appears to be none of the two divisions from Why3 standard library.

Another example is the fact that all types in Why3 are inhabited. Automated theorem provers typically make that assumption already. When it comes to proof assistants Coq and PVS, however, it is not granted for free. PVS provides a built-in mechanism for non-empty types, that we use readily. Coq, on the contrary, does not provide any such facility. We resort to type classes to ensure that all types we manipulate are inhabited. For instance, function `length` is translated into

```
Fixpoint length {a:Type} {a_WT:WhyType a} (l:(list a)) ...
```

where `WhyType a` is a type class that states that type `a` is inhabited and has decidable equality.

3.3 Proof Sessions

When a verification tool is built on top of a single automated prover, it is straightforward to replay a proof: one simply reruns the tool on the input files. The same applies for a proof that has been built interactively, say with Coq or KIV, and stored into files: it can be rechecked easily, in batch mode.

With Why3, the situation is somewhat different. Using the graphical user interface `why3ide`, the user interactively applies transformations and calls external theorem provers, until all goals are discharged. Calls to provers record

the maximal amount of CPU time and memory that is allocated to the prover. Calls to proof assistants record user-edited proof scripts. All that information is stored into a set of files called a *proof session*. This way, it can be reloaded on a subsequent call to `why3ide` or replayed in a batch mode with another tool, `why3replayer`. This is of particular interest when the user upgrades the version of one or several provers. A run of `why3replayer` then tells whether the proof session is still valid or must be updated. Proof sessions are the subject of another publication [9].

3.4 Examples

We now give some examples of the way goals are translated to be passed to theorem provers. Let us consider our running example in Fig. 11 and let us assume that we are targeting the SMT solver Alt-Ergo.

Algebraic Data Types. We must get rid of algebraic data types. Thus, `list 'a` is simply turned into some uninterpreted data type, together with uninterpreted function symbols for constructors

```
type 'a list
logic Nil : 'a list
logic Cons : 'a, 'a list → 'a list
```

and axioms (not shown here) to state that `Nil` and `Cons` are distinct and that `Cons` is injective. To get rid of pattern matching on type `list`, we also introduce a function symbol

```
logic match_list : 'a list, 'a1, 'a1 → 'a1
```

together with axioms stating that $\texttt{match\_list}(\texttt{Nil}, a, b) = a$ and that $\texttt{match\_list}(\texttt{Cons}(x, y), a, b) = b$. We proceed similarly for type `nat`.

Recursive Definitions. We must also get rid of the recursive definition of function `length`. It is turned into some uninterpreted function symbol together with two axioms:

```
logic length : 'a list → nat
axiom length_def : length(Nil : 'a list) = 0
axiom length_def1 : forall x:'a. forall x1:'a list.
  length(Cons(x, x1)) = S(length(x1))
```

Note that we are not making use of function `match_list` here, since it immediately reduces to simpler axioms.

Inductive Definitions. Finally, we must get rid of the inductive definition of predicate `even`. It is axiomatized as follows:

```
logic even : nat → prop
axiom Even0 : even(0)
axiom EvenS : forall m:nat. even(m) → even(S(S(m)))
axiom even_inversion :
  forall z:nat. even(z) →
    (z = 0 or exists m:nat. even(m) and z = S(S(m)))
```

This is incomplete, since we cannot capture the minimality of `even` in first-order logic. For instance, Alt-Ergo cannot prove the goal `forall x: nat. even x → not (even (S x))`, while we could use Coq to prove it. Anyway, Alt-Ergo easily manages to prove the goal

```
goal G: even(length(Cons(0, Cons(0, (Nil : nat list)))))
```

with suitable instantiations of lemmas `length_def`, `length_def1`, `Even0`, and `EvenS`.

Types. As we see on the example above, we did not have to encode types. Indeed, Alt-Ergo supports polymorphic types [8]. If we are instead targeting a prover that only supports simple types (*e.g.* Z3) or some untyped logic (*e.g.* SPASS), we have to encode the polymorphic types of Why3 in some way or another. For instance, on the following Why3 input

```
type list 'a
constant nil: list 'a
function length (list 'a) : int
axiom length_nil: length (nil: list 'a) = 0
goal G: length (nil: list int) = 0
```

the file that is passed to SPASS is the following:

```
fof(length_nil, axiom,
![A]: sort(int, length(sort(list(A),nil))) = sort(int,const_0)).

fof(g, conjecture,
sort(int, length(sort(list(int), nil))) = sort(int,const_0)).
```

Table 1. Comparing eight automated provers using Why3

prover	proved	max. time	avg. time
CVC3 (2.4.1)	2203	21.00	0.17
Alt-Ergo (0.95.1)	2202	29.73	0.16
CVC4 (1.0)	2071	12.00	0.09
Z3 (4.3.1)	1869	45.52	0.11
Yices (1.0.38)	1634	4.30	0.05
Vampire (0.6)	1375	27.72	0.51
E (1.6)	1303	19.73	0.41
Spass (3.7)	1185	23.78	0.52

Here `sort` is a binary function symbol that wraps terms with types. Types themselves are represented as regular terms, such as constant `int` or variable `A` above. The case of an SMT solver is more subtle, as we need to protect built-in types — such as integers, arrays, or reals — if we want to use built-in decision procedures [11].

4 Experimental Results

In this section, we use Why3 to run a small benchmark of 8 different automated provers. The experiment uses 83 proof sessions from our gallery, corresponding to logical theories or programs that were all successfully proved. This includes our solutions to several recent competitions in program verification (VSTTE 2012, FoVeOOS 2011, VSTTE 2012, FM 2012). The total number of subgoals is 2849.

For each subgoal that was discharged by at least one prover, we run the other 7 provers on that subgoal, with the same limit of CPU time that was given to the first one. Provers are run on an 8-cores 3.20GHz Intel Xeon with 24Gb of RAM. Each prover runs on a separate core, with a limit of 1Gb of memory. Results are given in Table 1. For each prover, we give the total number of goals proved, and the maximum/average running time per goal.

The purpose of that table is not really to compare provers, but rather to show the benefits of a collaborative use of several provers: if we were using CVC3 only, we would be left with 646 ($= 2849 - 2203$) unproved subgoals. Besides, it is worth pointing out that most of these goals involve arithmetic; yet provers with no support for arithmetic (E, SPASS, and Vampire, in that case) are able to discharge a large subset of the goals. This was rather unexpected and encourages us to increase our daily use of these provers and to improve the way we use them even further.

5 Related Work

There is actually very little in the literature regarding the extension of first-order logic with polymorphism. For instance, a classical textbook such as Manzano's [32] does not contain a single occurrence of the word 'polymorphism'. One the other side of the logical spectrum, rich logics such as the Calculus of Inductive Constructions or HOL do have polymorphism (even beyond rank-1) but are seldom interested in identifying their first-order fragments. Among the recent work on this subject, we can mention the work by Leino and Rümmer on Boogie 2's type system [28] and the definition of the TPTP TFF1 format by Blanchette and Paskevich [6].

There is more related work regarding the second part of this paper, as a lot has been done in the context of Isabelle's Sledgehammer tactic [34,12]. It translates Isabelle's logic to several external provers, using type encodings different from ours, ranging from mere type erasure (which is unsound, but Sledgehammer uses proof reconstruction[2]) to partial monomorphisation [5] (which is proved

[2] A significant difference between Sledgehammer and our work is that we do not perform any kind of proof reconstruction. Thus we have to keep to sound encodings.

incomplete [11] but seems efficient in practice anyway). Earlier, Hurd had already investigated encodings from higher-order to first-order logic [24].

6 Conclusion and Perspectives

We have presented a logic whose purpose is to provide a unified front-end to many existing theorem provers, either interactive or automated, and its implementation in Why3. We have designed this logic with the idea that specification must be as natural as possible and that tools should adapt themselves to proof obligations (and not the opposite). Using a wide range of theorem provers encourages this attitude.

One of the key features of our logic is polymorphism. It is defined and handled roughly the same way it is in tools Boogie, Sledgehammer, and Alt-Ergo. There is no competition, but rather a nice convergence. A contribution of this paper is to add the formalization of algebraic data types, recursive definitions, and inductive predicates on top of that.

Currently, Why3 is successfully used as a sub-component in various projects [23,33,4,14], as well as the vehicle for many non-trivial case studies in program verification (see for instance [21]). We also envision that verification environments that are currently built on top of a dedicated prover (*e.g.* B, KIV, SmallFoot) could also benefit from additional, external theorem provers. For instance, we are currently experimenting with the use of Why3 to discharge goals obtained from Atelier B [35], and the first results are promising.

We could still improve the way we are using theorem provers. For instance, we can observe that some tools that are based on a single automated prover — *e.g.* VCC, Dafny, or VeriFast — are able to carry out impressive case studies. It is clear that these tools are achieving a level of intimacy with the prover that is beyond that of Why3. We should learn from these tools and transpose relevant techniques to Why3. In particular, we our support of built-in theories.

Of course, it would be much simpler if we had native support for polymorphic types in provers. Alt-Ergo demonstrates that this is possible, and even simple to implement [8], yet this is currently the only automated prover with such a feature. We hope that TFF1 [6] will encourage some ATP developers to take the plunge.

Acknowledgments. I would like to thank Andrei Paskevich and Sylvain Conchon for fruitful discussions during the preparation of this paper. Some ideas behind the formalization in Sec. 2 already appear in Bobot and Paskevich's work [11]. The development of Why3 is joint work with François Bobot, Claude Marché, Guillaume Melquiond, and Andrei Paskevich.

References

1. Abrial, J.-R.: The B-Book, assigning programs to meaning. Cambridge University Press (1996)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
3. Barrett, C.W., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
4. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)
5. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 493–507. Springer, Heidelberg (2013)
6. Blanchette, J.C., Paskevich, A.: TFF1: The TPTP typed first-order form with rank-1 polymorphism. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 414–420. Springer, Heidelberg (2013)
7. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo automated theorem prover (2008), `http://alt-ergo.lri.fr/`
8. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing Polymorphism in SMT solvers. In: Barrett, C., de Moura, L. (eds.) SMT 2008: 6th International Workshop on Satisfiability Modulo. ACM International Conference Proceedings Series, vol. 367, pp. 1–5 (2008)
9. Bobot, F., Filliâtre, J.-C., Marché, C., Melquiond, G., Paskevich, A.: Preserving user proofs across specification changes. In: Cohen, E., Rybalchenko, A. (eds.) Verified Software: Theories, Tools, Experiments (5th International Conference VSTTE), Atherton, USA, May 2013. LNCS, Springer (2013)
10. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland, pp. 53–64 (August 2011)
11. Bobot, F., Paskevich, A.: Expressing Polymorphic Types in a Many-Sorted Language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 87–102. Springer, Heidelberg (2011)
12. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer, Heidelberg (2010)
13. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: An extensible automated theorem prover producing checkable proofs. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 151–165. Springer, Heidelberg (2007)
14. Comar, C., Kanig, J., Moy, Y.: Integrating formal program verification with testing. In: Proceedings of the Embedded Real Time Software and Systems Conference, ERTS^2 2012 (February 2012)
15. Couchot, J.-F., Lescuyer, S.: Handling polymorphism in automated deduction. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 263–278. Springer, Heidelberg (2007)
16. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. Transactions on Mathematical Software 37(1), 1–20 (2010)
17. de Moura, L., Bjørner, N.S.: Z3, an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)

18. de Moura, L., Dutertre, B.: Yices: An SMT Solver, http://yices.csl.sri.com/
19. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52, 365–473 (2005)
20. Filliâtre, J.-C.: Deductive software verification. International Journal on Software Tools for Technology Transfer (STTT) 13(5), 397–403 (2011)
21. Filliâtre, J.-C.: Verifying two lines of C with Why3: an exercise in program verification. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 83–97. Springer, Heidelberg (2012)
22. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013)
23. The Frama-C platform for static analysis of C programs (2008), http://www.frama-c.cea.fr/
24. Hurd, J.: An lcf-style interface between hol and first-order logic. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 134–138. Springer, Heidelberg (2002)
25. Jacobs, B., Piessens, F.: The VeriFast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven (August 2008)
26. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. Communications of the ACM 53(6), 107–115 (2010)
27. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
28. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010)
29. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
30. Leino, K.R.M.: Automating induction with an SMT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 315–331. Springer, Heidelberg (2012)
31. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
32. Manzano, M.: Extensions of first order logic. Cambridge University Press, New York (1996)
33. Marché, C., Paulin-Mohring, C., Urbain, X.: The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. Journal of Logic and Algebraic Programming 58(1–2), 89–106 (2004), http://krakatoa.lri.fr
34. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. Journal of Automated Reasoning 40, 35–60 (2008)
35. Mentré, D., Marché, C., Filliâtre, J.-C., Asuka, M.: Discharging proof obligations from Atelier B using multiple automated provers. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 238–251. Springer, Heidelberg (2012), http://hal.inria.fr/hal-00681781/en/
36. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)

37. Reif, W., Schnellhorn, G., Stenzel, K.: Proving system correctness with KIV 3.0. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, pp. 69–72. Springer, Heidelberg (1997)
38. Riazanov, A., Voronkov, A.: Vampire. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 292–296. Springer, Heidelberg (1999)
39. Schulte, W., Xia, S., Smans, J., Piessens, F.: A glimpse of a verifying C compiler, `http://www.cs.ru.nl/~tews/cv07/cv07-smans.pdf`
40. Schulz, S.: System description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 223–228. Springer, Heidelberg (2004)
41. The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.3 (2010), `http://coq.inria.fr`
42. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 140–145. Springer, Heidelberg (2009)

The Tree Width of Separation Logic with Recursive Definitions

Radu Iosif[1], Adam Rogalewicz[2], and Jiri Simacek[2]

[1] VERIMAG/CNRS, Grenoble, France
[2] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. Separation Logic is a widely used formalism for describing dynamically allocated linked data structures, such as lists, trees, etc. The decidability status of various fragments of the logic constitutes a long standing open problem. Current results report on techniques to decide satisfiability and validity of entailments for Separation Logic(s) over lists (possibly with data). In this paper we establish a more general decidability result. We prove that any Separation Logic formula using rather general recursively defined predicates is decidable for satisfiability, and moreover, entailments between such formulae are decidable for validity. These predicates are general enough to define (doubly-) linked lists, trees, and structures more general than trees, such as trees whose leaves are chained in a list. The decidability proofs are by reduction to decidability of Monadic Second Order Logic on graphs with bounded tree width.

1 Introduction

Separation Logic (SL) [17] is a general framework for describing dynamically allocated mutable data structures generated by programs that use pointers and low-level memory allocation primitives. The logics in this framework are used by an important number of academic (SPACE INVADER [1], SLEEK [16] and PREDATOR [9]), as well as industrial-scale (INFER [7]) tools for program verification and certification. These logics are used both externally, as property specification languages, or internally, as e.g., abstract domains for computing invariants, or for proving verification conditions. The main advantage of using SL when dealing with heap manipulating programs, is the ability to provide compositional proofs, based on the principle of *local reasoning* i.e., analyzing different sections (e.g., functions, threads, etc.) of the program, that work on disjoint parts of the global heap, and combining the analysis results a-posteriori.

The basic language of SL consists of two kinds of atomic propositions describing either (i) the empty heap, or (ii) a heap consisting of an allocated cell, connected via a separating conjunction primitive. Hence a basic SL formula can describe only a heap whose size is bounded by the size of the formula. The ability of describing unbounded data structures is provided by the use of *recursive definitions*. Figure 1 gives several common examples of recursive data structures definable in this framework.

The main difficulty that arises when using Separation Logic with Recursive Definitions (SLRD) to reason automatically about programs is that the logic, due to its expressiveness, does not have very nice decidability properties. Most dialects used in practice restrict the language (e.g., no quantifier alternation, the negation is used in a

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 21–38, 2013.

$$
\begin{array}{rcl}
list(hd,tl) & ::= & emp \wedge hd = tl \\
& | & \exists x.\ hd \mapsto x * list(x,tl) \\
dll(hd,p,tl) & ::= & emp \wedge hd = tl \\
& | & \exists x.\ hd \mapsto (x,p) * dll(x,hd,tl) \\
tree(root) & ::= & emp \wedge root = nil \\
& | & \exists l,r.\ root \mapsto (l,r) * tree(l) * tree(r) \\
tll(x,ll,lr) & ::= & x \mapsto (nil,nil,lr) \wedge x = ll \\
& | & \exists l,r,z.\ x \mapsto (l,r,nil) * tll(l,ll,z) \\
& & * tll(r,z,lr)
\end{array}
$$

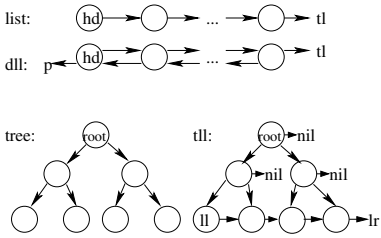


Fig. 1. Examples of recursive data structures definable in SLRD

very restricted ways, etc.) and the class of models over which the logic is interpreted (typically singly-linked lists, and slight variations thereof). In the same way, we apply several natural restrictions on the syntax of the recursive definitions, and define the fragment SLRD_{btw}, which guarantees that all models of a formula in the fragment have *bounded tree width*. Indeed, this ensures that the satisfiability and entailment problems in this fragment are decidable *without any restrictions on the type of the recursive data structures considered*.

In general, the techniques used in proving decidability of Separation Logic are either proof-based ([16,2]), or model-based ([5,8]). It is well-known that automata theory, through various automata-logics connections, provides a unifying framework for proving decidability of various logics, such as (W)SkS, Presburger Arithmetic or MSO over certain classes of graphs. In this paper we propose an automata-theoretic approach consisting of two ingredients. First, SLRD_{btw} formulae are translated into equivalent Monadic Second Order (MSO) formulae over graphs. Second, we show that the models of SLRD_{btw} formulae have the *bounded tree width* property, which provides a decidability result by reduction to the satisfiability problem for MSO interpreted over graphs of bounded tree width [18], and ultimately, to the emptiness problem of tree automata.

Related Work. The literature on defining decidable logics for describing mutable data structures is rather extensive. Initially, first-order logic with transitive closure of one function symbol was introduced in [11] with a follow-up logic of reachability on complex data structures, in [19]. The decision procedures for these logics are based on reductions to the decidability of MSO over finite trees. Along the same lines, the logic PALE [15] goes beyond trees, in defining trees with edges described by regular routing expressions, whose decidability is still a consequence of the decidability of MSO over trees. More recently, the CSL logic [4] uses first-order logic with reachability (along multiple selectors) in combination with arithmetic theories to reason about shape, path lengths and data within heap structures. Their decidability proof is based on a small model property, and the algorithm is enumerative. In the same spirit, the STRAND logic [14] combines MSO over graphs, with quantified data theories, and provides decidable fragments using a reduction to MSO over graphs of bounded tree width.

On what concerns SLRD [17], the first (proof-theoretic) decidability result on a restricted fragment defining only singly-linked lists was reported in [2], which describe a coNP algorithm. The full basic SL without recursive definitions, but with the magic

wand operator was found to be undecidable when interpreted *in any memory model* [6]. Recently, the entailment problem for SLRD over lists has been reduced to graph homomorphism in [8], and can be solved in PTIME. This method has been extended to reason nested and overlaid lists in [10]. The logic SLRD_{btw}, presented in this paper is, to the best of our knowledge, the first decidable SL that can define structures more general than lists and trees, such as e.g. trees with parent pointers and linked leaves.

2 Preliminaries

For a finite set S, we denote by $\|S\|$ its cardinality. We sometimes denote sets and sequences of variables as $\mathbf{x}$, the distinction being clear from the context. If $\mathbf{x}$ denotes a sequence, $(\mathbf{x})_i$ denotes its i-th element. For a partial function $f : A \rightharpoonup B$, and $\bot \notin B$, we denote $f(x) = \bot$ the fact that f is undefined at some point $x \in A$. By $f[a \leftarrow b]$ we denote the function λx . if $x = a$ then b else $f(x)$. The domain of f is denoted $dom(f) = \{x \in A \mid f(x) \neq \bot\}$, and the image of f is denoted as $img(f) = \{y \in B \mid \exists x \in A \,.\, f(x) = y\}$. By $f : A \rightharpoonup_{fin} B$ we denote any partial function whose domain is finite. Given two partial functions f, g defined on disjoint domains, we denote by $f \oplus g$ their union.

Stores, Heaps and States. We consider $PVar = \{u, v, w, \ldots\}$ to be a countable infinite set of *pointer variables* and $Loc = \{l, m, n, \ldots\}$ to be a countable infinite set of *memory locations*. Let $nil \in PVar$ be a designated variable, $null \in Loc$ be a designated location, and $Sel = \{1, \ldots, \mathcal{S}\}$, for some given $\mathcal{S} > 0$, be a finite set of natural numbers, called *selectors* in the following.

Definition 1. *A* state *is a pair $\langle s, h\rangle$ where $s : PVar \rightharpoonup Loc$ is a partial function mapping pointer variables into locations such that $s(nil) = null$, and $h : Loc \rightharpoonup_{fin} Sel \rightharpoonup_{fin} Loc$ is a finite partial function such that (i) $null \notin dom(h)$ and (ii) for all $\ell \in dom(h)$ there exist $k \in Sel$ such that $(h(\ell))(k) \neq \bot$.*

Given a state $S = \langle s, h\rangle$, s is called the *store* and h the *heap*. For any $k \in Sel$, we write $h_k(\ell)$ instead of $(h(\ell))(k)$, and $\ell \xrightarrow{k} \ell'$ for $h_k(\ell) = \ell'$. We sometimes call a triple $\ell \xrightarrow{k} \ell'$ an *edge*, and k is called a *selector*. Let $Img(h) = \bigcup_{\ell \in Loc} img(h(\ell))$ be the set of locations which are destinations of some selector edge in h. A location $\ell \in Loc$ is said to be *allocated* in $\langle s, h\rangle$ if $\ell \in dom(h)$ (i.e. it is the source of an edge), and *dangling* in $\langle s, h\rangle$ if $\ell \in [img(s) \cup Img(h)] \setminus dom(h)$, i.e., it is either referenced by a store variable, or reachable from an allocated location in the heap, but it is not allocated in the heap itself. The set $loc(S) = img(s) \cup dom(h) \cup Img(h)$ is the set of all locations either allocated or referenced in a state $S = \langle s, h\rangle$.

Trees. Let Σ be a finite label alphabet, and $\mathbb{N}^*$ be the set of sequences of natural numbers. Let $\varepsilon \in \mathbb{N}^*$ denote the empty sequence, and $p.q$ denote the concatenation of two sequences $p, q \in \mathbb{N}^*$. A *tree* t over Σ is a finite partial function $t : \mathbb{N}^* \rightharpoonup_{fin} \Sigma$, such that $dom(t)$ is a finite prefix-closed subset of $\mathbb{N}^*$, and for each $p \in dom(t)$ and $i \in \mathbb{N}$, we have: $t(p.i) \neq \bot \Rightarrow \forall 0 \leq j < i \,.\, t(p.j) \neq \bot$. Given two positions $p, q \in dom(t)$, we say

that q is the i-th successor (child) of p if $q = p.i$, for $i \in \mathbb{N}$. Also q is a successor of p, or equivalently, p is the parent of q, denoted $p = parent(q)$ if $q = p.i$, for some $i \in \mathbb{N}$.

We will sometimes denote by $\mathcal{D}(t) = \{-1, 0, \ldots, N\}$ the *direction alphabet* of t, where $N = \max\{i \in \mathbb{N} \mid p.i \in dom(t)\}$. The concatenation of positions is defined over $\mathcal{D}(t)$ with the convention that $p.(-1) = q$ if and only if $p = q.i$ for some $i \in \mathbb{N}$. We denote $\mathcal{D}_+(t) = \mathcal{D}(t) \setminus \{-1\}$. A *path* in t, from p_1 to p_k, is a sequence $p_1, p_2, \ldots, p_k \in dom(t)$ of pairwise distinct positions, such that either $p_i = parent(p_{i+1})$ or $p_{i+1} = parent(p_i)$, for all $1 \leq i < k$. Notice that a path in the tree can also link sibling nodes, not just ancestors to their descendants, or viceversa. However, a path may not visit the same tree position twice.

Tree Width. A state (Def. 1) can be seen as a directed graph, whose nodes are locations, and whose edges are defined by the selector relation. Some nodes are labeled by program variables (*PVar*) and all edges are labeled by selectors (*Sel*). The notion of tree width is then easily adapted from generic labeled graphs to states. Intuitively, the tree width of a state (graph) measures the similarity of the state to a tree.

Definition 2. *Let $S = \langle s, h \rangle$ be a state. A* tree decomposition *of S is a tree $t : \mathbb{N}^* \rightharpoonup_{fin} 2^{loc(S)}$, labeled with sets of locations from $loc(S)$, with the following properties:*

1. *$loc(S) = \bigcup_{p \in dom(t)} t(p)$, the tree covers the locations of S*
2. *for each edge $l_1 \xrightarrow{s} l_2$ in S, there exists $p \in dom(t)$ such that $l_1, l_2 \in t(p)$*
3. *for each $p, q, r \in dom(t)$, if q is on a path from p to r in t, then $t(p) \cap t(r) \subseteq t(q)$*

The width of the decomposition is $w(t) = \max_{p \in dom(t)}\{\|t(p)\| - 1\}$. The tree width of S is $tw(S) = \min\{w(t) \mid t \text{ is a tree decomposition of } S\}$.

A set of states is said to have *bounded tree width* if there exists a constant $k \geq 0$ such that $tw(S) \leq k$, for any state S in the set. Figure 2 gives an example of a graph (left) and a possible tree decomposition (right).

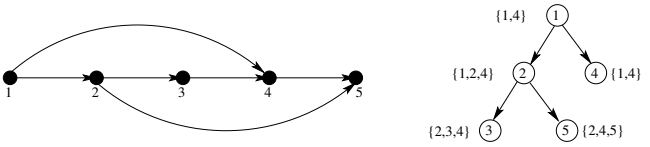


Fig. 2. A graph and a possible tree decomposition of width 2

2.1 Syntax and Semantics of Monadic Second Order Logic

Monadic second-order logic (MSO) on states is a straightforward adaptation of MSO on labeled graphs [13]. As usual, we denote first-order variables, ranging over locations, by $x, y, \ldots$, and second-order variables, ranging over sets of locations, by $X, Y, \ldots$. The set of logical MSO variables is denoted by $LVar_{mso}$, where $PVar \cap LVar_{mso} = \emptyset$.

We emphasize here the distinction between the logical variables $LVar_{mso}$ and the pointer variables $PVar$: the former may occur within the scope of first and second order quantifiers, whereas the latter play the role of symbolic constants (function symbols of zero arity). For the rest of this paper, a logical variable is said to be free if it does not occur within the scope of a quantifier. By writing $\varphi(\mathbf{x})$, for an MSO formula φ, and a set of logical variables $\mathbf{x}$, we mean that all free variables of φ are in $\mathbf{x}$.

The syntax of MSO is defined below:

$$u \in PVar;\ x, X \in LVar_{mso};\ k \in \mathbb{N}$$
$$\varphi ::= x = y \mid var_u(x) \mid edge_k(x,y) \mid null(x) \mid X(x) \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x.\varphi \mid \exists X.\varphi$$

The semantics of MSO on states is given by the relation $S, \iota, \nu \models_{mso} \varphi$, where $S = \langle s, h \rangle$ is a state, $\iota : \{x, y, z, \ldots\} \rightharpoonup_{fin} Loc$ is an interpretation of the first order variables, and $\nu : \{X, Y, Z, \ldots\} \rightharpoonup_{fin} 2^{Loc}$ is an interpretation of the second order variables. If $S, \iota, \nu \models_{mso} \varphi$ for all interpretations $\iota : \{x, y, z, \ldots\} \rightharpoonup_{fin} Loc$ and $\nu : \{X, Y, Z, \ldots\} \rightharpoonup_{fin} 2^{Loc}$, then we say that S is a *model* of φ, denoted $S \models_{mso} \varphi$. We use the standard MSO semantics [18], with the following interpretations of the vertex and edge labels:

$$\begin{aligned} S, \iota, \nu \models_{mso} null(x) &\iff \iota(x) = nil \\ S, \iota, \nu \models_{mso} var_u(x) &\iff s(u) = \iota(x) \\ S, \iota, \nu \models_{mso} edge_k(x,y) &\iff h_k(\iota(x)) = \iota(y) \end{aligned}$$

The *satisfiability problem* for MSO asks, given a formula φ, whether there exists a state S such that $S \models_{mso} \varphi$. This problem is, in general, undecidable. However, one can show its decidability on a restricted class of models. The theorem below is a slight variation of a classical result in (MSO-definable) graph theory [18]. For space reasons, all proofs are given in [12].

Theorem 1. *Let $k \geq 0$ be an integer constant, and φ be an MSO formula. The problem asking if there exists a state S such that $tw(S) \leq k$ and $S \models_{mso} \varphi$ is decidable.*

2.2 Syntax and Semantics of Separation Logic

Separation Logic (SL) [17] uses only a set of first order logical variables, denoted as $LVar_{sl}$, ranging over locations. We suppose that $LVar_{sl} \cap PVar = \emptyset$ and $LVar_{sl} \cap LVar_{mso} = \emptyset$. Let Var_{sl} denote the set $PVar \cup LVar_{sl}$. A formula is said to be *closed* if it does not contain logical variables which are not under the scope of a quantifier. By writing $\varphi(\mathbf{x})$ for an SL formula φ and a set of logical variables $\mathbf{x}$, we mean that all free variables of φ are in $\mathbf{x}$.

Basic Formulae. The syntax of basic formula is given below:

$$\alpha \in Var_{sl} \setminus \{nil\};\ \beta \in Var_{sl};\ x \in LVar_{sl}$$
$$\pi ::= \alpha = \beta \mid \alpha \neq \beta \mid \pi_1 \wedge \pi_2$$
$$\sigma ::= emp \mid \alpha \mapsto (\beta_1, \ldots, \beta_n) \mid \sigma_1 * \sigma_2 \text{ , for some } n > 0$$
$$\varphi ::= \pi \wedge \sigma \mid \exists x\ .\ \varphi$$

A formula of the form $\bigwedge_{i=1}^{n} \alpha_i = \beta_i \wedge \bigwedge_{j=1}^{m} \alpha_j \neq \beta_j$ defined by π in the syntax above is said to be *pure*. If Π is a pure formula, let Π^* denote its *closure*, i.e., the equivalent pure formula obtained by the exhaustive application of the reflexivity, symmetry, and transitivity axioms of equality. A formula of the form $\bigstar_{i=1}^{k} \alpha_i \mapsto (\beta_{i,1}, \ldots, \beta_{i,n})$ defined by σ in the syntax above is said to be *spatial*. The atomic proposition *emp* denotes the empty spatial conjunction. For a spatial formula Σ, let $|\Sigma|$ be the total number of variable occurrences in Σ, e.g. $|emp| = 0$, $|\alpha \mapsto (\beta_1, \ldots, \beta_n)| = n+1$, etc.

The semantics of a basic formula φ is given by the relation $S, \iota \models_{sl} \varphi$ where $S = \langle s, h \rangle$ is a state, and $\iota : LVar_{sl} \rightharpoonup_{fin} Loc$ is an interpretation of logical variables from φ. For a closed formula φ, we denote by $S \models_{sl} \varphi$ the fact that S is a *model* of φ.

$$
\begin{array}{lll}
S, \iota \models_{sl} emp & \iff & dom(h) = \emptyset \\
S, \iota \models_{sl} \alpha \mapsto (\beta_1, \ldots, \beta_n) & \iff & h = \{\langle (s \oplus \iota)(\alpha), \lambda i \,.\, \text{if } i \leq n \text{ then } (s \oplus \iota)(\beta_i) \text{ else } \bot \rangle\} \\
S, \iota \models_{sl} \varphi_1 * \varphi_2 & \iff & S_1, \iota \models_{sl} \varphi_1 \text{ and } S_2, \iota \models_{sl} \varphi_2 \text{ where } S_1 \uplus S_2 = S
\end{array}
$$

The semantics of $=$, $\neq$, $\wedge$, and $\exists$ is classical. Here, the notation $S_1 \uplus S_2 = S$ means that S is the union of two states $S_1 = \langle s_1, h_1 \rangle$ and $S_2 = \langle s_2, h_2 \rangle$ whose stacks agree on the evaluation of common program variables ($\forall \alpha \in PVar \,.\, s_1(\alpha) \neq \bot \wedge s_2(\alpha) \neq \bot \Rightarrow s_1(\alpha) = s_2(\alpha)$), and whose heaps have disjoint domains ($dom(h_1) \cap dom(h_2) = \emptyset$) i.e., $S = \langle s_1 \cup s_2, h_1 \oplus h_2 \rangle$. Note that we adopt here the *strict semantics*, in which a points-to relation $\alpha \mapsto (\beta_1, \ldots, \beta_n)$ holds in a state consisting of a single cell pointed to by α, with exactly n outgoing edges towards dangling locations pointed to by $\beta_1, \ldots, \beta_n$, and the empty heap is specified by *emp*.

Every basic formula φ is equivalent to an existentially quantified pair $\Sigma \wedge \Pi$ where Σ is a spatial formula and Π is a pure formula. Given a basic formula φ, one can define its spatial (Σ) and pure (Π) parts uniquely, up to equivalence. A variable $\alpha \in Var$ is said to be *allocated in* φ if and only if $\alpha \mapsto (\ldots)$ occurs in Σ. It is easy to check that an allocated variable may not refer to a dangling location in any model of φ. A variable β is *referenced* if and only if $\alpha \mapsto (\ldots, \beta, \ldots)$ occurs in Σ for some variable α. For a basic formula $\varphi \equiv \Sigma \wedge \Pi$, the *size of* φ is defined as $|\varphi| = |\Sigma|$.

Lemma 1. *Let $\varphi(\mathbf{x})$ be a basic SL formula, $S = \langle s, h \rangle$ be a state, and $\iota : LVar_{sl} \rightharpoonup_{fin} Loc$ be an interpretation, such that $S, \iota \models_{sl} \varphi(\mathbf{x})$. Then $tw(S) \leq \max(|\varphi|, \|PVar\|)$.*

Recursive Definitions. A system $\mathcal{P}$ of *recursive definitions* is of the form:

$$
\begin{array}{c}
P_1(x_{1,1}, \ldots, x_{1,n_1}) ::= \mid_{j=1}^{m_1} R_{1,j}(x_{1,1}, \ldots, x_{1,n_1}) \\
\ldots \\
P_k(x_{k,1}, \ldots, x_{k,n_k}) ::= \mid_{j=1}^{m_k} R_{k,j}(x_{k,1}, \ldots, x_{k,n_k})
\end{array}
$$

where $P_1, \ldots, P_k$ are called *predicates*, $x_{i,1}, \ldots, x_{i,n_i}$ are called *parameters*, and the formulae $R_{i,j}$ are called the *rules* of P_i. Concretely, a rule $R_{i,j}$ is of the form $R_{i,j}(\mathbf{x}) \equiv \exists \mathbf{z} \,.\, \Sigma * P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_m}(\mathbf{y}_m) \wedge \Pi$, where Σ is a spatial SL formula over variables $\mathbf{x} \cup \mathbf{z}$, called the *head* of $R_{i,j}$, $\langle P_{i_1}(\mathbf{y}_1), \ldots, P_{i_m}(\mathbf{y}_m) \rangle$ is an ordered sequence of *predicate occurrences*, called the *tail* of $R_{i,j}$ (we assume w.l.o.g. that $\mathbf{x} \cap \mathbf{z} = \emptyset$, and that $\mathbf{y}_k \subseteq \mathbf{x} \cup \mathbf{z}$, for all $k = 1, \ldots, m$), Π is a pure formula over variables $\mathbf{x} \cup \mathbf{z}$.

Without losing generality, we assume that all variables occurring in a rule of a recursive definition system are logical variables from $LVar_{sl}$ – pointer variables can be passed as parameters at the top level. We subsequently denote $head(R_{i,j}) \equiv \Sigma$, $tail(R_{i,j}) \equiv \langle P_{i_k}(\mathbf{y}_k)\rangle_{k=1}^{m}$ and $pure(R_{i,j}) \equiv \Pi$, for each rule $R_{i,j}$. Rules with empty tail are called *base cases*. For each rule $R_{i,j}$ let $\|R_{i,j}\|^{var} = \|\mathbf{z}\| + \|\mathbf{x}\|$ be the number of variables, both existentially quantified and parameters, that occur in $R_{i,j}$. We denote by $\|\mathcal{P}\|^{var} = \max\{\|R_{i.j}\|^{var} \mid 1 \leq i \leq k,\ 1 \leq j \leq m_i\}$ the maximum such number, among all rules in $\mathcal{P}$. We also denote by $\mathcal{D}(\mathcal{P}) = \{-1, 0, \ldots, \max\{|tail(R_{i,j})| \mid 1 \leq i \leq k,\ 1 \leq j \leq m_i\} - 1\}$ the *direction alphabet* of $\mathcal{P}$.

Example. The predicate tll describes a data structure called a *tree with parent pointers and linked leaves* (see Fig. 3(b)). The data structure is composed of a binary tree in which each internal node points to left and right children, and also to its parent node. In addition, the leaves of the tree are kept in a singly-linked list, according to the order in which they appear on the frontier (left to right).

$$\begin{array}{rll} tll(x,p,leaf_l,leaf_r) ::= & x \mapsto (nil,nil,p,leaf_r) \wedge x = leaf_l & (R_1) \\ \mid & \exists l,r,z.\ x \mapsto (l,r,p,nil) * tll(l,x,leaf_l,z) * tll(r,x,z,leaf_r) & (R_2) \end{array}$$

The base case rule (R_1) allocates leaf nodes. The internal nodes of the tree are allocated by the rule (R_2), where the ttl predicate occurs twice, first for the left subtree, and second for the right subtree. □

Definition 3. *Given a system of recursive definitions* $\mathcal{P} = \left\{P_i ::= |_{j=1}^{m_i} R_{i,j}\right\}_{i=1}^{n}$, *an* unfolding tree of $\mathcal{P}$ rooted at i *is a finite tree* t *such that:*

1. *each node of* t *is labeled by a single rule of the system* $\mathcal{P}$,
2. *the root of* t *is labeled with a rule of* P_i,
3. *nodes labeled with base case rules have no successors, and*
4. *if a node* u *of* t *is labeled with a rule whose tail is* $P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_m}(\mathbf{y}_m)$, *then the children of* u *form the ordered sequence* $v_1, \ldots, v_m$ *where* v_j *is labeled with one of the rules of* P_{i_j} *for all* $j = 1, \ldots, m$.

Remarks. Notice that the recursive predicate $P(x) ::= \exists y\ .\ x \mapsto y * P(y)$ does not have finite unfolding trees. However, in general a system of recursive predicates may have infinitely many finite unfolding trees. □

In the following, we denote by $\mathcal{T}_i(\mathcal{P})$ the set of unfolding trees of $\mathcal{P}$ rooted at i. An unfolding tree $t \in \mathcal{T}_i(\mathcal{P})$ corresponds to a basic formula of separation logic ϕ_t, called the *characteristic formula* of t, and defined in what follows. For a set of tree positions $P \subseteq \mathbb{N}^*$, we denote $LVar^P = \{x^p \mid x \in LVar,\ p \in P\}$. For a tree position $p \in \mathbb{N}^*$ and a rule R, we denote by R^p the rule obtained by replacing every variable occurrence x in R by x^p. For each position $p \in dom(t)$, we define a formula ϕ_t^p, by induction on the structure of the subtree of t rooted at p:

- if p is a leaf labeled with a base case rule R, then $\phi_t^p \equiv R^p$
- if p has successors $p.1, \ldots, p.m$, and the label of p is the recursive rule $R(\mathbf{x}) \equiv \exists \mathbf{z}\ .\ head(R) * \bigstar_{j=1}^{m} P_{i_j}(\mathbf{y}_j) \wedge pure(R)$, then:

$$\phi_t^p(\mathbf{x}^p) \equiv \exists \mathbf{z}^p\ .\ head(R^p) * \bigstar_{j=1}^{m} [\exists \mathbf{x}_{i_j}^{p.i}\ .\ \phi_t^{p.i}(\mathbf{x}_{i_j}^{p.i}) \wedge \mathbf{y}_j^p = \mathbf{x}_{i_j}^{p.i}] \wedge pure(R^p)$$

In the rest of the paper, we write ϕ_t for ϕ_t^{ε}. Notice that ϕ_t is defined using the set of logical variables $LVar^{dom(t)}$, instead of $LVar$. However the definition of SL semantics from the previous carries over naturally to this case.

Example. (cont'd) Fig. 3(a) presents an unfolding tree for the *tll* predicate given in the previous example. The characteristic formula of each node in the tree can be obtained by composing the formulae labeling the children of the node with the formula labeling the node. The characteristic formula of the tree is the formula of its root. □

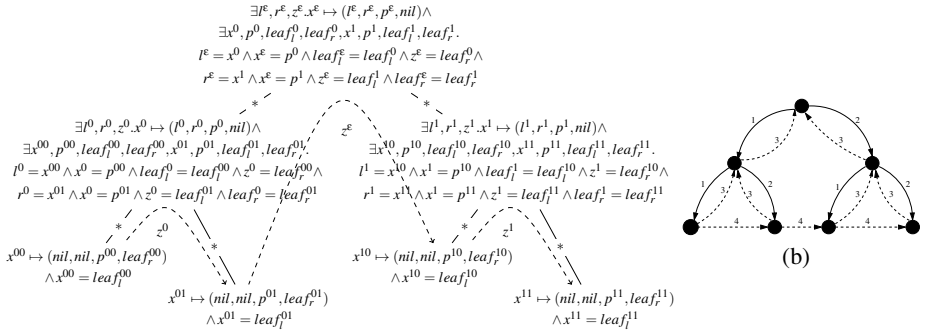


Fig. 3. (a) An unfolding tree for tll predicate and (b) a model of the corresponding formula

Given a system of recursive definitions $\mathcal{P} = \left\{P_i ::= |_{j=1}^{m_i} R_{i,j}\right\}_{i=1}^{n}$, the semantics of a recursive predicate P_i is defined as follows:

$$S,\mathfrak{u} \models_{sl} P_i(x_{i,1},\ldots,x_{i,n_i}) \iff S,\mathfrak{u}^{\varepsilon} \models_{sl} \phi_t(x_{i,1}^{\varepsilon},\ldots,x_{i,n_i}^{\varepsilon}), \text{ for some } t \in \mathcal{T}_i(\mathcal{P}) \qquad (1)$$

where $\mathfrak{u}^{\varepsilon}(x_{i,j}^{\varepsilon}) \stackrel{def}{=} \mathfrak{u}(x_{i,j})$ for all $j = 1,\ldots,n_i$.

Remark. Since the recursive predicate $P(x) ::= \exists y \,.\, x \mapsto y * P(y)$ does not have finite unfolding trees, the formula $\exists x.P(x)$ is unsatisfiable. □

Top Level Formulae. We are now ready to introduce the fragment of *Separation Logic with Recursive Definitions* (SLRD). A formula in this fragment is an existentially quantified formula of the following form: $\exists \mathbf{z} \,.\, \varphi * P_{i_1} * \ldots * P_{i_n}$, where φ is a basic formula, and P_{i_j} are occurrences of recursive predicates, with free variables in $PVar \cup \mathbf{z}$. The semantics of an SLRD formula is defined in the obvious way, from the semantics of the basic fragment, and that of the recursive predicates.

Example. The following SLRD formulae, with $PVar = \{root, head\}$, describe both the set of binary trees with parent pointer and linked leaves, rooted at *root*, with the leaves

linked into a list pointed to by *head*. The difference is that φ_1 describes also a tree containing only a single allocated location:

$\varphi_1 \equiv tll(root, nil, head, nil)$
$\varphi_2 \equiv \exists l, r, x. root \mapsto (l, r, nil, nil) * tll(l, root, head, x) * tll(r, root, x, nil)$ □

We are interested in solving two problems on SLRD formulae, namely *satisfiability* and *entailment*. The satisfiability problem asks, given a closed SLRD formula φ, whether there exists a state S such that $S \models_{sl} \varphi$. The entailment problem asks, given two closed SLRD formulae φ_1 and φ_2, whether for all states S, $S \models_{sl} \varphi_1$ implies $S \models_{sl} \varphi_2$. This is denoted also as $\varphi_1 \models_{sl} \varphi_2$. For instance, in the previous example we have $\varphi_2 \models_{sl} \varphi_1$, but not $\varphi_1 \models_{sl} \varphi_2$.

In general, it is possible to reduce an entailment problem $\varphi_1 \models \varphi_2$ to satisfiability of the formula $\varphi_1 \wedge \neg\varphi_2$. In our case, however, this is not possible directly, because SLRD is not closed under negation. The decision procedures for satisfiability and entailment is the subject of the rest of this paper.

3 Decidability of Satisfiability and Entailment in SLRD

The decision procedure for the satisfiability and entailment in SLRD is based on two ingredients. First, we show that, under certain natural restrictions on the system of recursive predicates, which define a fragment of SLRD, called SLRD_{btw}, all states that are models of SLRD_{btw} formulae have *bounded tree width* (Def. 2). These restrictions are as follows:

1. *Progress*: each rule allocates exactly one variable
2. *Connectivity*: there is at least one selector edge between the variable allocated by a rule and the variable allocated by each of its children in the unfolding tree
3. *Establishment*: all existentially quantified variables in a recursive rule are eventually allocated

Second, we provide a *translation of SLRD_{btw} formulae into equivalent MSO formulae*, and rely on the fact that satisfiability of MSO is decidable on classes of states with bounded tree width.

3.1 A Decidable Subset of SLRD

At this point we define the SLRD_{btw} fragment formally, by defining the three restrictions above. The *progress* condition (1) asks that, for each rule R in the system of recursive definitions, we have $head(R) \equiv \alpha \mapsto (\beta_1, \ldots, \beta_n)$, for some variables $\alpha, \beta_1, \ldots, \beta_n \in Var_{sl}$. The intuition between this restriction is reflected by the following example.

Example. Consider the following system of recursive definitions:

$$ls(x, y) ::= x \mapsto y \mid \exists z, t \,.\, x \mapsto (z, nil) * t \mapsto (nil, y) * ls(z, t)$$

The predicate $ls(x, y)$ defines the set of structures $\{x(\xrightarrow{1})^n z \mapsto t(\xrightarrow{2})^n y \mid n \geq 0\}$, which clearly cannot be defined in MSO. □

The *connectivity* condition (2) is defined below:

Definition 4. *A rule R of a system of recursive definitions, such that $head(R) \equiv \alpha \mapsto (\beta_1, \ldots, \beta_n)$ and $tail(R) \equiv \langle P_{i_1}(\mathbf{y}_1), \ldots, P_{i_m}(\mathbf{y}_m)\rangle$, $m \geq 1$, is said to be* connected *if and only if the following hold:*

- *for each $j = 1, \ldots, m$, $(\mathbf{y}_j)_s = \beta'$, for some $1 \leq s \leq n_{i_j}$, where n_{i_j} is the number of parameters of P_{i_j}*
- *$\beta_t = \beta'$ occurs in $pure(R)^*$, for some $1 \leq t \leq n$*
- *the s-th parameter $x_{i_j,s}$ of P_{i_j} is allocated in the heads of all rules of P_{i_j}.*

In this case we say that between rule R and any rule Q of P_{i_j}, there is a *local edge*, labeled by *selector* t. $\mathcal{F}(R, j, Q) \subseteq Sel$ denotes the set of all such selectors. If all rules of $\mathcal{P}$ are connected, we say that $\mathcal{P}$ is connected.

Example. The following recursive rule, from the previous tll predicate, is connected:

$$\exists l, r, z \;.\; x \mapsto (l, r, p, nil) * tll(l, x, leaf_l, z) * tll(r, x, z, leaf_r) \quad (R_2)$$

R_2 is connected because the variable l is referenced in R_2 and it is passed as the first parameter to tll in the first recursive call to tll. Moreover, the first parameter (x) is allocated by all rules of tll. R_2 is connected, for similar reasons. We have $\mathcal{F}(R_2, 1, R_2) = \{1\}$ and $\mathcal{F}(R_2, 2, R_2) = \{2\}$. □

The *establishment* condition (3) is formally defined below.

Definition 5. *Let $P(x_1, \ldots, x_n) = |_{j=1}^{m} R_j(x_1, \ldots, x_n)$ be a predicate in a recursive system of definitions. We say that a parameter x_i, for some $i = 1, \ldots, n$ is* allocated *in P if and only if, for all $j = 1, \ldots, m$:*

- *either x_i is allocated in $head(R_j)$, or*
- *(i) $tail(R_j) = \langle P_{i_1}(\mathbf{y}_1), \ldots, P_{i_k}(\mathbf{y}_k)\rangle$, (ii) $(\mathbf{y}_\ell)_s = x_i$ occurs in $pure(R_j)^*$, for some $\ell = 1, \ldots, k$, and (iii) the s-th parameter of P_{i_ℓ} is allocated in P_{i_ℓ}*

A system of recursive definitions is said to be established *if and only if every existentially quantified variable is allocated.*

Example. Let $llextra(x) ::= x \mapsto (nil, nil) \mid \exists n, e.\; x \mapsto (n, e) * llextra(n)$ be a recursive definition system, and let $\phi ::= llextra(head)$, where $head \in PVar$. The models of the formula ϕ are singly-linked lists, where in all locations of the heap, the first selector points to the next location in the list, and the second selector is dangling i.e., it can point to any location in the heap. These dangling selectors may form a squared grid of arbitrary size, which is a model of the formula ϕ. However, the set of squared grids does not have bounded tree width [18]. The problem arises due to the existentially quantified variables e which are never allocated. □

Given a system $\mathcal{P}$ of recursive definitions, one can effectively check whether it is established, by guessing, for each predicate $P_i(x_{i,1}, \ldots, x_{i,n_i})$ of $\mathcal{P}$, the minimal set of parameters which are allocated in P_i, and verify this guess inductively[1]. Then, once the minimal set of allocated parameters is determined for each predicate, one can check whether every existentially quantified variable is eventually allocated.

[1] For efficiency, a least fixpoint iteration can be used instead of a non-deterministic guess.

Lemma 2. *Let $\mathcal{P} = \{P_i ::= |_{j=1}^{m_i} R_{ij}(x_{i,1}, \ldots, x_{i,n_i})\}_{i=1}^{k}$ be a established system of recursive definitions, and $S = \langle s, h \rangle$ be a state, such that $S, \iota \models_{sl} P_i(x_{i,1}, \ldots, x_{i,n_i})$ for some interpretation $\iota : LVar_{sl} \rightharpoonup_{fin} Loc$ and some $1 \leq i \leq k$. Then $tw(S) \leq \|\mathcal{P}\|^{var}$.*

The result of the previous lemma extends to an arbitrary top-level formula:

Theorem 2. *Let $\mathcal{P} = \{P_i ::= |_{j=1}^{m_i} R_{ij}(x_{i,1}, \ldots, x_{i,n_i})\}_{i=1}^{k}$ be a established system of recursive definitions, and $S = \langle s, h \rangle$ be a state, such that $S \models_{sl} \exists \mathbf{z} \,.\, \varphi(\mathbf{y}_0) * P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_n}(\mathbf{y}_n)$, where φ is a basic SL formula, and P_{i_j} are predicates of $\mathcal{P}$, and $\mathbf{y}_i \subseteq \mathbf{z}$, for all $i = 0, 1, \ldots, n$. Then $tw(S) \leq \max(\|\mathbf{z}\|, |\varphi|, \|PVar\|, \|\mathcal{P}\|^{var})$.*

4 From SLRD$_{btw}$ to MSO

This section describes the translation of a SL formula using recursively defined predicates into an MSO formula. We denote by $\Pi(X_0, \ldots, X_i, X)$ the fact that $X_0, \ldots, X_i$ is a partition of X, and by $\Sigma(x, X)$ the fact that X is a singleton with x as the only element.

4.1 Converting Basic SL Formulae to MSO

For every SL logical variable $x \in LVar_{sl}$ we assume the existence of an MSO logical variable $\overline{x} \in LVar_{mso}$, which is used to replace x in the translation. For every program variable $u \in PVar \setminus \{nil\}$ we assume the existence of a logical variable $\overline{x_u} \in LVar_{mso}$. The special variable $nil \in LVar_{sl}$ is translated into $\overline{x_{nil}} \in LVar_{mso}$ (with the associated MSO constraint $null(\overline{x_{nil}})$). In general, for any pointer or logical variable $\alpha \in Var_{sl}$, we denote by $\overline{\alpha}$, the logical MSO variable corresponding to it.

The translation of a pure SL formula $\alpha = \beta$, $\alpha \neq \beta$, $\pi_1 \wedge \pi_2$ is $\overline{\alpha} = \overline{\beta}$, $\neg(\overline{\alpha} = \overline{\beta})$, $\overline{\pi_1} \wedge \overline{\pi_2}$, respectively, where $\overline{\pi}(\overline{\alpha_1}, \ldots, \overline{\alpha_k})$ is the translation of $\pi(\alpha_1, \ldots, \alpha_k)$. Spatial SL formulae $\sigma(\alpha_1, \ldots, \alpha_k)$ are translated into MSO formulae $\overline{\sigma}(\overline{\alpha_1}, \ldots, \overline{\alpha_k}, X)$, where X is used for the set of locations allocated in σ. The fact that X actually denotes the domain of the heap, is ensured by the following MSO constraint:

$$Heap(X) \equiv \forall x \bigvee_{i=1}^{\|Sel\|} (\exists y \,.\, edge_i(x, y)) \leftrightarrow X(x)$$

The translation of basic spatial formulae is defined by induction on their structure:

$$\begin{array}{ll}
\overline{emp}(X) & \equiv \forall x \,.\, \neg X(x) \\
\overline{(\alpha \mapsto (\beta_1, \ldots, \beta_n))}(X) & \equiv \Sigma(\overline{\alpha}, X) \wedge \bigwedge_{i=1}^{n} edge_i(\overline{\alpha}, \overline{\beta_i}) \wedge \bigwedge_{i=n+1}^{\|Sel\|} \forall x \,.\, \neg edge_i(\overline{\alpha}, x) \\
\overline{(\sigma_1 * \sigma_2)}(X) & \equiv \exists Y \exists Z \,.\, \overline{\sigma_1}(Y) \wedge \overline{\sigma_2}(Z) \wedge \Pi(Y, Z, X)
\end{array}$$

The translation of a closed basic SL formula φ in MSO is defined as $\exists X \,.\, \overline{\varphi}(X)$, where $\overline{\varphi}(X)$ is defined as $\overline{(\pi \wedge \sigma)}(X) \equiv \overline{\pi} \wedge \overline{\sigma}(X)$, and $\overline{(\exists x \,.\, \varphi_1)}(X) \equiv \exists \overline{x} \,.\, \overline{\varphi_1}(X)$. The following lemma proves that the MSO translation of a basic SL formula defines the same set of models as the original SL formula.

Lemma 3. *For any state $S = \langle s,h\rangle$, any interpretation $\iota : LVar_{sl} \rightharpoonup_{fin} Loc$, and any basic SL formula φ, we have $S,\iota \models_{sl} \varphi$ if and only if $S,\overline{\iota},\nu[X \leftarrow dom(h)] \models_{mso} \overline{\varphi}(X) \wedge Heap(X)$, where $\overline{\iota} : LVar_{mso} \rightharpoonup_{fin} Loc$ is an interpretation of first order variables, such that $\overline{\iota}(x_u) = s(u)$, for all $u \in PVar$, and $\overline{\iota}(\overline{x}) = \iota(x)$, for all $x \in LVar_{sl}$, and $\nu : LVar_{mso} \rightharpoonup_{fin} 2^{Loc}$ is any interpretation of second-order variables.*

4.2 States and Backbones

The rest of this section is concerned with the MSO definition of states that are models of recursive SL formulae, i.e. formulae involving recursively defined predicates. The main idea behind this encoding is that any part of a state which is the model of a recursive predicate can be decomposed into a tree-like structure, called the *backbone*, and a set of edges between the nodes in this tree. Intuitively, the backbone is a spanning tree that uses only *local edges*. For instance, in the state depicted in Fig. 3(b), the local edges are drawn in solid lines.

Let $P_k(x_1,\ldots,x_n)$ be a recursively defined predicate of a system $\mathcal{P}$, and $S,\iota \models_{sl} P_k(x_1,\ldots,x_n)$, for some state $S = \langle s,h\rangle$ and some interpretation $\iota : LVar_{sl} \to Loc$. Then $S,\iota \models_{sl} \phi_t$, where $t \in \mathcal{T}_k(\mathcal{P})$ is an unfolding tree, ϕ_t is its characteristic formula, and $\mu : dom(t) \to dom(h)$ is the bijective tree that describes the allocation of nodes in the heap by rules labeling the unfolding tree. Recall that the direction alphabet of the system $\mathcal{P}$ is $\mathcal{D}(\mathcal{P}) = \{-1,0,\ldots,N-1\}$, where N is the maximum number of predicate occurrences within some rule of $\mathcal{P}$, and denote $\mathcal{D}_+(\mathcal{P}) = \mathcal{D}(\mathcal{P}) \setminus \{-1\}$. For each rule R_{ij} in $\mathcal{P}$ and each direction $d \in \mathcal{D}(\mathcal{P})$, we introduce a second order variable X_{ij}^d to denote the set of locations ℓ such that (i) $t(\mu^{-1}(\ell)) \equiv R_{ij}$ and (ii) $\mu^{-1}(\ell)$ is a d-th child, if $d \geq 0$, or $\mu^{-1}(\ell)$ is the root of t, if $d = -1$. Let $\overrightarrow{\mathbf{X}}$ be the sequence of X_{ij}^k variables, enumerated in some order. We use the following shorthands:

$$X_{ij}(x) \equiv \bigvee_{k \in \mathcal{D}(\mathcal{P})} X_{ij}^k(x) \qquad X_i(x) \equiv \bigvee_{1 \leq j \leq m_i} X_{ij}(x) \qquad X_i^k(x) \equiv \bigvee_{1 \leq j \leq m_i} X_{ij}^k(x)$$

to denote, respectively, locations that are allocated by a rule R_{ij} (X_{ij}), by a recursive predicate P_i (X_i), or by a predicate P_i, who are mapped to a k-th child (or to the root, if $k = -1$) in the unfolding tree of $\mathcal{P}$, rooted at i (X_i^k).

In order to characterize the backbone of a state, one must first define the local edges:

$$local\_edge_{i,j,p,q}^d(x,y) \equiv \bigwedge_{s \in \mathcal{F}(R_{i,j},d,R_{pq})} edge_s(x,y)$$

for all $d \in \mathcal{D}_+(\mathcal{P})$. Here $\mathcal{F}(R_{ij},d,R_{pq})$ is the set of forward local selectors for direction d, which was defined previously – notice that the set of local edges depends on the source and destination rules R_{ij} and R_{pq}, that label the corresponding nodes in the unfolding tree, respectively. The following predicate ensures that these labels are used correctly, and define the successor functions in the unfolding tree:

$$succ_d(x,y,\overrightarrow{\mathbf{X}}) \equiv \bigvee_{\substack{1 \leq i,p \leq M \\ 1 \leq j \leq m_i \\ 1 \leq q \leq m_p}} X_{ij}(x) \wedge X_{pq}^k(y) \wedge local\_edge_{i,j,p,q}^d(x,y)$$

for all $d \in \mathcal{D}_+(\mathcal{P})$. The definition of the backbone of a recursive predicate P_i in MSO follows tightly the definition of the unfolding tree of $\mathcal{P}$ rooted at i (Def. 3):

$$backbone_i(r, \overrightarrow{\mathbf{X}}, T) \equiv tree(r, \overrightarrow{\mathbf{X}}, T) \wedge X_i^{-1}(r) \wedge succ\_labels(\overrightarrow{\mathbf{X}})$$

where $tree(r, \overrightarrow{\mathbf{X}}, T)$ defines a tree[2] with domain T, rooted at r, with successor functions defined by $succ_0, \ldots, succ_{N-1}$, and $succ\_labels$ ensures that the labeling of each tree position (with rules of $\mathcal{P}$) is consistent with the definition of $\mathcal{P}$:

$$succ\_labels(\overrightarrow{\mathbf{X}}) \equiv \bigwedge_{\substack{1 \leq i \leq M \\ 1 \leq j \leq m_i}} X_{ij}(x) \rightarrow \bigwedge_{d=0}^{r_{ij}-1} \exists y \,.\, X_{k_d}^d(y) \wedge succ_d(x, y, \overrightarrow{\mathbf{X}}) \wedge \forall y \,.\, \bigwedge_{p=s_{ij}+1}^{\|Sel\|} \neg edge_p(x, y)$$

where we suppose that, for each rule R_{ij} of $\mathcal{P}$, we have $head(R_{ij}) \equiv \alpha \mapsto (\beta_1, \ldots, \beta_{s_{ij}})$ and $tail(R_{ij}) = \langle P_{k_1}, \ldots, P_{k_{r_{ij}}} \rangle$, for some $r_{ij} \geq 0$, and some indexing $k_1, \ldots, k_{r_{ij}}$ of predicate occurrences within R_{ij}. The last conjunct ensures that a location allocated in R_{ij} does not have more outgoing edges than specified by $head(R_{ij})$. This condition is needed, since, unlike SL, the semantics of MSO does not impose strictness conditions on the number of outgoing edges.

4.3 Inner Edges

An edge between two locations is said to be *inner* if both locations are allocated in the heap. Let μ be the bijective tree defined in Sec. 4.2. The existence of an edge $\ell \xrightarrow{k} \ell'$ in S, between two arbitrary locations $\ell, \ell' \in dom(h)$, is the consequence of:

1. a basic points-to formula $\alpha \mapsto (\beta_1, \ldots, \beta_k, \ldots, \beta_n)$ that occurs in $\mu(\ell)$
2. a basic points-to formula $\gamma \mapsto (\ldots)$ that occurs in $\mu(\ell')$
3. a path $\mu(\ell) = p_1, p_2, \ldots, p_{m-1}, p_m = \mu(\ell')$ in t, such that the equalities $\beta_k^{p_1} = \delta_2^{p_2} = \ldots = \delta_{m-1}^{p_{m-1}} = \gamma^{p_m}$ are all logical consequences of ϕ_t, for some tree positions $p_2, \ldots, p_{m-1} \in dom(t)$ and some variables $\delta_2, \ldots, \delta_{m-1} \in LVar_{sl}$.

Notice that the above conditions hold only for inner edges. The (corner) case of edges leading to dangling locations is dealt with in [12].

Example. The existence of the edge from tree position 00 to 01 in Fig. 3(b), is a consequence of the following: (1) $x^{00} \mapsto (nil, nil, p^{00}, leaf_r^{00})$, (2) $x^{01} \mapsto (nil, nil, p^{01}, leaf_r^{01})$, and (3) $leaf_r^{00} = z^0 = leaf_l^{01} = x^{01}$. The reason for other dashed edges is similar. □

The main idea here is to encode in MSO the existence of such paths, in the unfolding tree, between the source and the destination of an edge, and use this encoding to define the edges. To this end, we use a special class of tree automata, called *tree-walking automata* (TWA) to recognize paths corresponding to sequences of equalities occurring within characteristic formulae of unfolding trees.

[2] For space reasons this definition can be found in [12].

Tree Walking Automata. Given a set of tree directions $\mathcal{D} = \{-1, 0, \ldots, N\}$ for some $N \geq 0$, a tree-walking automaton[3], is a tuple $A = (\Sigma, Q, q_i, q_f, \Delta)$ where Σ is a set of tree node labels, Q is a set of states, $q_i, q_f \in Q$ are the initial and final states, and $\Delta : Q \times (\Sigma \cup \{root\}) \times (\Sigma \cup \{?\}) \to 2^{Q \times (\mathcal{D} \cup \{\varepsilon\})}$ is the (non-deterministic) transition function. A configuration of A is a pair $\langle p, q \rangle$, where $p \in \mathcal{D}^*$ is a tree position, and $q \in Q$ is a state. A run of A over a Σ-labeled tree t is a sequence of configurations $\langle p_1, q_1 \rangle, \ldots, \langle p_n, q_n \rangle$, with $p_1, \ldots, p_n \in dom(t)$, such that for all $i = 1, \ldots, n-1$, we have $p_{i+1} = p_i.k$, where either:

1. $p_i \neq \varepsilon$ and $(q_{i+1}, k) \in \Delta(q_i, t(p_i), t(p_i.(-1)))$, for $k \in \mathcal{D} \cup \{\varepsilon\}$
2. $p_i = \varepsilon$ and $(q_{i+1}, k) \in \Delta(q_i, \sigma, ?)$, for $\sigma \in \{t(p_i) \cup root\}$ and $k \in \mathcal{D} \cup \{\varepsilon\}$

The run is said to be *accepting* if $q_1 = q_i$, $p_1 = \varepsilon$ and $q_n = q_f$.

Routing Automata. For a system of recursive definitions $\mathcal{P} = \left\{ P_i(x_{i,1}, \ldots, x_{i,n_i}) ::= \mid_{j=1}^{m_i} R_{ij}(x_{i,1}, \ldots, x_{i,n_i}) \right\}_{i=1}^{k}$, we define the TWA $A_{\mathcal{P}} = (\Sigma_{\mathcal{P}}, Q_{\mathcal{P}}, q_i, q_f, \Delta_{\mathcal{P}})$, where $\Sigma_{\mathcal{P}} = \{R_{ij}^k \mid 1 \leq i \leq k,\ 1 \leq j \leq m_i,\ k \in \mathcal{D}(\mathcal{P})\}$, $Q_{\mathcal{P}} = \{q_x^{var} \mid x \in LVar_{sl}\} \cup \{q_s^{sel} \mid s \in Sel\} \cup \{q_i, q_f\}$. The transition function $\Delta_{\mathcal{P}}$ is defined as follows:

1. $(q_i, k), (q_s^{sel}, \varepsilon) \in \Delta(q_i, \sigma, \tau)$ for all $k \in \mathcal{D}_+(\mathcal{P})$, all $s \in Sel$ and all $\sigma \in \Sigma_{\mathcal{P}} \cup \{root\}$, $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ i.e., the automaton first moves downwards chosing random directions, while in q_i, then changes to q_s^{sel} for some non-deterministically chosen selector s.
2. $(q_{\beta_s}^{var}, \varepsilon) \in \Delta(q_s^{sel}, R_{ij}^k, \tau)$ and $(q_f, \varepsilon) \in \Delta(q_\alpha^{var}, R_{ij}^k, \tau)$ for all $k \in \mathcal{D}(\mathcal{P})$ and $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ if and only if $head(R_{ij}) \equiv \alpha \mapsto (\beta_1, \ldots, \beta_s, \ldots, \beta_m)$, for some $m > 0$ i.e., when in q_s^{sel}, the automaton starts tracking the destination β_s of the selector s through the tree. The automaton enters the final state when the tracked variable α is allocated.
3. for all $k \in \mathcal{D}_+(\mathcal{P})$, all $\ell \in \mathcal{D}(\mathcal{P})$ and all rules $R_{\ell q}$ of $P_\ell(x_{\ell,1}, \ldots, x_{\ell,n_\ell})$, we have $(q_{x_{\ell,j}}^{var}, k) \in \Delta(q_{y_j}^{var}, R_{ij}^l, \tau)$, for all $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$, and $(q_{y_j}^{var}, -1) \in \Delta(q_{x_{\ell,j}}^{var}, R_{\ell q}^k, R_{ij}^l)$ if and only if $tail(R_{ij})_k \equiv P_\ell(y_1, \ldots, y_{n_\ell})$ i.e., the automaton moves down along the k-th direction tracking $x_{\ell,j}$ instead of y_j, when the predicate $P_\ell(\mathbf{y})$ occurs on the k-th position in R_{ij}. Symmetrically, the automaton can also move up tracking y_j instead of $x_{\ell,j}$, in the same conditions.
4. $(q_\beta^{var}, \varepsilon) \in \Delta(q_\alpha^{var}, R_{ij}^k, \tau)$ for all $k \in \mathcal{D}(\mathcal{P})$ and all $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ if and only if $\alpha = \beta$ occurs in $pure(R_{ij})$ i.e., the automaton switches from tracking α to tracking β when the equality between the two variables occurs in R_{ij}, while keeping the same position in the tree.

The following lemma formalizes the correctness of the TWA construction:

Lemma 4. *Given a system of recursive definitions $\mathcal{P}$, and an unfolding tree $t \in \mathcal{T}_i(\mathcal{P})$ of $\mathcal{P}$, rooted at i, for any $x, y \in LVar_{sl}$ and $p, r \in dom(t)$, we have $\models_{sl} \phi_t \to x^p = y^r$ if and only if $A_{\mathcal{P}}$ has a run from $\langle p, q_x^{var} \rangle$ to $\langle r, q_y^{var} \rangle$ over t, where ϕ_t is the characteristic formula of t.*

[3] This notion of tree-walking automaton is a slightly modified but equivalent to the one in [3]. We give the translation of TWA into the original definition in [12].

To the routing automaton $A_{\mathcal{P}}$ corresponds the MSO formula $\Phi_{A_{\mathcal{P}}}(r, \overrightarrow{\mathbf{X}}, T, \overrightarrow{\mathbf{Y}})$, where r maps to the root of the unfolding tree, $\overrightarrow{\mathbf{X}}$ is the sequence of second order variables X_{ij}^k defined previously, T maps to the domain of the tree, and $\overrightarrow{\mathbf{Y}}$ is a sequence of second-order variables X_q, one for each state $q \in Q_{\mathcal{P}}$. We denote by Y_s^{sel} and Y_f the variables from $\overrightarrow{\mathbf{Y}}$ that correspond to the states q_S^{sel} and q_f, for all $s \in Sel$, respectively. For space reasons, the definition of $\Phi_{A_{\mathcal{P}}}$ is given in [12]. With this notation, we define:

$$inner\_edges(r, \overrightarrow{\mathbf{X}}, T) \equiv \forall x \forall y \bigwedge_{s \in Sel} \exists \overrightarrow{\mathbf{Y}} \,.\, \Phi_{A_{\mathcal{P}}}(r, \overrightarrow{\mathbf{X}}, T, \overrightarrow{\mathbf{Y}}) \wedge Y_s^{sel}(x) \wedge Y_f(y) \rightarrow edge_s(x, y)$$

4.4 Double Allocation

In order to translate the definition of a recursively defined SL predicate $P(x_1, \ldots, x_n)$ into an MSO formula $\overline{P}$, that captures the models of P, we need to introduce a sanity condition, imposing that recursive predicates which establish equalities between variables allocated at different positions in the unfolding tree, are unsatisfiable, due to the semantics of the separating conjunction of SL, which implicitly conjoins all local formulae of an unfolding tree. A double allocation occurs in the unfolding tree t if and only if there exist two distinct positions $p, q \in dom(t)$ and:

1. a basic points-to formula $\alpha \mapsto (\ldots)$ occurring in $t(p)$
2. a basic points-to formula $\beta \mapsto (\ldots)$ occurring in $t(q)$
3. a path $p = p_1, \ldots, p_m = q$ in t, such that the equalities $\alpha^p = \gamma_2^{p_2} = \ldots = \gamma_{m-1}^{p_{m-1}} = \beta^q$ are all logical consequences of ϕ_t, for some tree positions $p_2, \ldots, p_{m-1} \in dom(t)$ and some variables $\gamma_2, \ldots, \gamma_{m-1} \in LVar_{sl}$

The cases of double allocation can be recognized using a routing automaton $B_{\mathcal{P}} = (\Sigma_{\mathcal{P}}, Q'_{\mathcal{P}}, q_i, q_f, \Delta'_{\mathcal{P}})$, whose states $Q'_{\mathcal{P}} = \{q_x^{var} \mid x \in LVar_{sl}\} \cup \{q_0, q_i, q_f\}$ and transitions $\Delta'_{\mathcal{P}}$ differ from $A_{\mathcal{P}}$ only in the following rules:

- $(q_0, \varepsilon) \in \Delta(q_i, \sigma, \tau)$ for all $\sigma \in \Sigma_{\mathcal{P}} \cup \{root\}$ and all $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$, i.e. after non-deterministically chosing a position in the tree, the automaton enters a designated state q_0, which occurs only once in each run.
- $(q_\alpha^{var}, \varepsilon) \in \Delta(q_0, R_{ij}^k, \tau)$ for all $k \in \mathcal{D}(\mathcal{P})$ and all $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ if and only if $head(R_{ij}) = \alpha \mapsto (\ldots)$, while in the designated state q_0, the automaton starts tracking the variable α, which is allocated at that position.

This routing automaton has a run over t, which labels one position by q_0 and a distinct one by q_f if and only if two positions in t allocate the same location. Notice that $B_{\mathcal{P}}$ has always a trivial run that starts and ends in the same position – since each position $p \in dom(t)$ allocates a variable α, and $\langle q_i, \varepsilon \rangle, \ldots, \langle q_0, p \rangle, \langle q_\alpha^{var}, p \rangle, \langle q_f, p \rangle$ is a valid run of $B_{\mathcal{P}}$. The predicate system has no double allocation if and only if these are the only possible runs of $B_{\mathcal{P}}$.

The existence of a run of $B_{\mathcal{P}}$ is captured by an MSO formula $\Phi_{B_{\mathcal{P}}}(r, \overrightarrow{\mathbf{X}}, T, \overrightarrow{\mathbf{Y}})$, where r maps to the root of the unfolding tree, $\overrightarrow{\mathbf{X}}$ is the sequence of second order variables X_{ij}^k defined previously, T maps to the domain of the tree, and $\overrightarrow{\mathbf{Y}}$ is the sequence of

second-order variables Y_q, taken in some order, each of which maps to the set of tree positions visited by the automaton while in state $q \in Q'_{\mathcal{P}}$ – we denote by Y_0 and Y_f the variables from $\overrightarrow{\mathbf{Y}}$ that correspond to the states q_0 and q_f, respectively. Finally, we define the constraint: $no\_double\_alloc(r, \overrightarrow{\mathbf{X}}, T) \equiv \forall \overrightarrow{\mathbf{Y}} \ . \ \Phi_{B_{\mathcal{P}}}(r, \overrightarrow{\mathbf{X}}, T, \overrightarrow{\mathbf{Y}}) \rightarrow Y_0 = Y_f$

4.5 Handling Parameters

The last issue to be dealt with is the role of the actual parameters passed to a recursively defined predicate $P_i(x_{i,1}, \ldots, x_{i,k})$ of $\mathcal{P}$, in a top-level formula. Then, for each parameter $x_{i,j}$ of P_i and each unfolding tree $t \in \mathcal{T}_i(\mathcal{P})$, there exists a path $\varepsilon = p_1, \ldots, p_m \in dom(t)$ and variables $\alpha_1, \ldots, \alpha_m \in LVar_{sl}$ such that $x_{i,j} \equiv \alpha_1$ and $\alpha_\ell^{p_\ell} = \alpha_{\ell+1}^{p_{\ell+1}}$ is a consequence of ϕ_t, for all $\ell = 1, \ldots, m-1$. Subsequently, there are three (not necessarily disjoint) possibilities:

1. $head(t(p_m)) \equiv \alpha_m \mapsto (\ldots)$, i.e. α_m is allocated
2. $head(t(p_m)) \equiv \beta \mapsto (\gamma_1, \ldots, \gamma_p, \ldots, \gamma_\ell)$, and $\alpha_m \equiv \gamma_p$, i.e. α_m is referenced
3. $\alpha_m \equiv x_{i,q}$ and $p_m = \varepsilon$, for some $1 \leq q \leq k$, i.e. α_m is another parameter $x_{i,q}$

Again, we use slightly modified routing automata (one for each of the case above) $C^{i,j}_{\mathcal{P},c} = (\Sigma_{\mathcal{P}}, Q''_{\mathcal{P}}, q_i, q_f, \Delta^{i,j}_c)$ for the cases $c = 1,2,3$, respectively. Here $Q''_{\mathcal{P}} = \{q_x^{var} \mid x \in LVar_{sl}\} \cup \{q_s^{sel} \mid s \in Sel\} \cup \{q^{i,a} \mid 1 \leq a \leq k\} \cup \{q_i, q_f\}$ and $\Delta^{i,j}_c$, $c = 1,2,3$ differ from the transitions of $A_{\mathcal{P}}$ in the following:

- $(q^{i,j}, \varepsilon) \in \Delta^{i,j}_x(q_i, root, ?)$, i.e. the automaton marks the root of the tree with a designated state $q^{i,j}$, that occurs only once on each run
- $(q^{var}_{x_{i,j}}, \varepsilon) \in \Delta^{i,j}_x(q^{i,j}, R^{-1}_{ik}, ?)$, for each rule R_{ik} of P_i, i.e. the automaton starts tracking the parameter variable $x_{i,j}$ beginning with the root of the tree
- $(q_f, \varepsilon) \in \Delta^{i,j}_1(q^{var}_\alpha, R^k_{ij}, \tau)$, for all $k \in \mathcal{D}(\mathcal{P})$, $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ iff $head(R_{ij}) \equiv \alpha \mapsto (\ldots)$ is the final rule for $C^{i,j}_{\mathcal{P},1}$
- $(q^{sel}_s, \varepsilon) \in \Delta^{i,j}_2(q^{var}_\gamma, R^k_{ij}, \tau)$, for all $k \in \mathcal{D}(\mathcal{P})$ and $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ iff $head(R_{ij}) \equiv \alpha \mapsto (\beta_1, \ldots, \beta_s, \ldots, \beta_n)$ and $\gamma \equiv \beta_s$ i.e., q^{sel}_s is reached in the second case, when the tracked variable is referenced. After that, $C^{i,j}_{\mathcal{P},2}$ moves to the final state i.e., $(q_f, \varepsilon) \in \Delta^{i,j}_2(q^{sel}_s, \sigma, \tau)$ for all $s \in Sel$, all $\sigma \in \Sigma_{\mathcal{P}} \cup \{root\}$ and $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$
- $(q^{i,a}, \varepsilon) \in \Delta^{i,j}_3(q^{var}_{x_{i,a}}, root, ?)$ and $(q_f, \varepsilon) \in \Delta^{i,j}_3(q_{i,a}, root, ?)$, for each $1 \leq a \leq k$ and $a \neq j$ i.e., are the final moves for $C^{i,j}_{\mathcal{P},3}$

The outcome of this construction are MSO formulae $\Phi_{C^{i,j}_{\mathcal{P},c}}(r, \overrightarrow{\mathbf{X}}, T, \overrightarrow{\mathbf{Y}})$, for $c = 1,2,3$, where r maps to the root of the unfolding tree, respectively, $\overrightarrow{\mathbf{X}}$ is the sequence of second order variables X^k_{ij} defined previously, T maps to the domain of the tree, and $\overrightarrow{\mathbf{Y}}$ is the sequence of second order variables corresponding to states of $Q''_{\mathcal{P}}$ – we denote by $Y_f, Y^{i,a}, Y^{sel}_s \in \overrightarrow{\mathbf{Y}}$ the variables corresponding to the states q_f, $q^{i,a}$, and q^{sel}_s, respectively. The parameter $x_{i,j}$ of P_i is assigned by the following MSO constraints:

$$param^1_{i,j}(r,\overrightarrow{\mathbf{X}},T) \equiv \exists \overrightarrow{\mathbf{Y}} \,.\, \Phi_{C^{i,j}_{\mathcal{P},1}} \wedge Y^{i,j}_0(\overline{x}_{i,j}) \wedge \forall y \,.\, Y_f(y) \rightarrow \overline{x}_{i,j} = y$$
$$param^2_{i,j}(r,\overrightarrow{\mathbf{X}},T) \equiv \exists \overrightarrow{\mathbf{Y}} \,.\, \Phi_{C^{i,j}_{\mathcal{P},2}} \wedge Y^{i,j}_0(\overline{x}_{i,j}) \wedge \bigwedge_{s\in Sel} \forall y \,.\, Y^{sel}_s(y) \rightarrow edge_s(y,\overline{x}_{i,j})$$
$$param^3_{i,j}(r,\overrightarrow{\mathbf{X}},T) \equiv \exists \overrightarrow{\mathbf{Y}} \,.\, \Phi_{C^{i,j}_{\mathcal{P},3}} \wedge Y^{i,j}_0(\overline{x}_{i,j}) \wedge \bigwedge_{1\leq a\leq k} \forall y \,.\, Y^{i,a}(y) \rightarrow \overline{x}_{i,j} = \overline{x}_{i,a}$$

where $\overline{x}_{i,j}$ is the first-order MSO variable corresponding to the SL parameter $x_{i,j}$. Finally, the constraint $param_{i,j}$ is conjunction of the $param^c_{i,j}$, $c = 1,2,3$ formulae.

4.6 Translating Top Level SLRD$_{btw}$ Formulae to MSO

We define the MSO formula corresponding to a predicate $P_i(x_{i,1},\ldots,x_{i,n_i})$, of a system of recursive definitions $\mathcal{P} = \{P_1,\ldots,P_n\}$:

$$\overline{P_i}(\overline{x}_{i,1},\ldots,\overline{x}_{i,n_i},T) \equiv \exists r \exists \overrightarrow{\mathbf{X}} \,.\, backbone_i(r,\overrightarrow{\mathbf{X}},T) \wedge inner\_edges(r,\overrightarrow{\mathbf{X}},T) \wedge no\_double\_alloc(r,\overrightarrow{\mathbf{X}},T) \wedge \bigwedge_{1\leq j\leq n_i} param_{i,j}(r,\overrightarrow{\mathbf{X}},T)$$

The following lemma is needed to establish the correctness of our construction.

Lemma 5. *For any state $S = \langle s,h\rangle$, any interpretation $\iota : LVar_{sl} \rightarrow_{fin} Loc$, and any recursively defined predicate $P_i(x_1,\ldots,x_n)$, we have $S,\iota \models_{sl} P_i(x_1,\ldots,x_n)$ if and only if $S,\overline{\iota},\nu[T \leftarrow dom(h)] \models_{mso} \overline{P_i}(\overline{x_1},\ldots,\overline{x_k},T) \wedge Heap(T)$, where $\overline{\iota} : LVar_{mso} \rightharpoonup_{fin} Loc$ is an interpretation of first order variables, such that $\overline{\iota}(x_u) = s(u)$, for all $u \in PVar$, and $\overline{\iota}(\overline{x}) = \iota(x)$, for all $x \in LVar_{sl}$, and $\nu : LVar_{mso} \rightharpoonup_{fin} 2^{Loc}$ is any interpretation of second-order variables.*

Recall that a top level SLRD$_{btw}$ formula is of the form: $\varphi \equiv \exists \mathbf{z} \,.\, \phi(\mathbf{y}_0) * P_{i_1}(\mathbf{y}_1) * \ldots P_{i_k}(\mathbf{y}_k)$, where $1 \leq i_1,\ldots,i_k \leq n$, and $\mathbf{y}_j \subseteq \mathbf{z}$, for all $j = 0,1,\ldots,k$. We define the MSO formula:

$$\overline{\varphi}(X) \equiv \exists \overline{\mathbf{z}} \exists X_{0,\ldots,k} \,.\, \overline{\phi}(\overline{\mathbf{y}_0},X_0) \wedge \overline{P_{i_1}}(\overline{\mathbf{y}_1},X_1) \wedge \ldots \wedge \overline{P_{i_k}}(\overline{\mathbf{y}_k},X_k) \wedge \Pi(X_0,X_1,\ldots,X_k,X)$$

Theorem 3. *For any state S and any closed SLRD$_{btw}$ formula φ we have that $S \models_{sl} \varphi$ if and only if $S \models_{mso} \exists X \,.\, \overline{\varphi}(X) \wedge Heap(X)$.*

Theorem 2 and the above theorem prove decidability of satisfiability and entailment problems for SLRD$_{btw}$, by reduction to MSO over states of bounded tree width.

5 Conclusions and Future Work

We defined a fragment of Separation Logic with Recursive Definitions, capable of describing general unbounded mutable data structures, such as trees with parent pointers and linked leaves. The logic is shown to be decidable for satisfiability and entailment, by reduction to MSO over graphs of bounded tree width. We conjecture that the complexity of the decision problems for this logic is elementary, and plan to compute tight upper bounds, in the near future.

Acknowledgement. This work was supported by the Czech Science Foundation (project P103/10/0306) and French National Research Agency (project VERIDYC ANR-09-SEGI-016). We also acknowledge Tomáš Vojnar, Lukáš Holík and the anonymous reviewers for their valuable comments.

References

1. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)
3. Bojańczyk, M.: Tree-walking automata. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 1–2. Springer, Heidelberg (2008)
4. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: A logic-based framework for reasoning about composite data structures. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 178–195. Springer, Heidelberg (2009)
5. Bozga, M., Iosif, R., Perarnau, S.: Quantitative separation logic and programs with lists. J. Autom. Reasoning 45(2), 131–156 (2010)
6. Brotherston, J., Kanovich, M.: Undecidability of propositional separation logic and its neighbours. In: Proceedings of the 2010 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, pp. 130–139 (2010)
7. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 459–465. Springer, Heidelberg (2011)
8. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 235–249. Springer, Heidelberg (2011)
9. Dudka, K., Peringer, P., Vojnar, T.: Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer, Heidelberg (2011)
10. Enea, C., Saveluc, V., Sighireanu, M.: Compositional invariant checking for overlaid and nested linked lists. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 129–148. Springer, Heidelberg (2013)
11. Immerman, N., Rabinovich, A., Reps, T., Sagiv, M., Yorsh, G.: The boundary between decidability and undecidability for transitive-closure logics. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 160–174. Springer, Heidelberg (2004)
12. Iosif, R., Rogalewicz, A., Simacek, J.: The tree width of separation logic with recursive definitions. CoRR abs/1301.5139 (2013)
13. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: Proc. of POPL 2011. ACM (2011)
14. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: Proc. of POPL 2011 (2011)
15. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Proc. of PLDI 2001 (June 2001)
16. Nguyen, H.H., Chin, W.-N.: Enhancing program verification with lemmas. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 355–369. Springer, Heidelberg (2008)
17. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proc. of LICS 2002. IEEE CS Press (2002)
18. Seese, D.: The structure of models of decidable monadic theories of graphs. Annals of Pure and Applied Logic 53(2), 169–195 (1991)
19. Yorsh, G., Rabinovich, A., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 94–110. Springer, Heidelberg (2006)

Hierarchic Superposition with Weak Abstraction

Peter Baumgartner[1] and Uwe Waldmann[2]

[1] NICTA* and Australian National University, Canberra, Australia
Peter.Baumgartner@nicta.com.au
[2] MPI für Informatik, Saarbrücken, Germany
uwe@mpi-inf.mpg.de

Abstract. Many applications of automated deduction require reasoning in first-order logic modulo background theories, in particular some form of integer arithmetic. A major unsolved research challenge is to design theorem provers that are "reasonably complete" even in the presence of free function symbols ranging into a background theory sort. The hierarchic superposition calculus of Bachmair, Ganzinger, and Waldmann already supports such symbols, but, as we demonstrate, not optimally. This paper aims to rectify the situation by introducing a novel form of clause abstraction, a core component in the hierarchic superposition calculus for transforming clauses into a form needed for internal operation. We argue for the benefits of the resulting calculus and provide a new completeness result for the fragment where all background-sorted terms are ground.

1 Introduction

Many applications of automated deduction require reasoning modulo background theories, in particular some form of integer arithmetic. Developing corresponding automated reasoning systems that are also able to deal with quantified formulas has recently been an active area of research. One major line of research is concerned with extending (SMT-based) solvers [15] for the quantifier-free case by instantiation heuristics for quantifiers [10,11, e. g.]. Another line of research is concerned with adding black-box reasoners for specific background theories to first-order automated reasoning methods (resolution [4,12,1], sequent calculi [17], instantiation methods [9,5,6], etc). In both cases, a major unsolved research challenge is to provide reasoning support that is "reasonably complete" in practice, so that the systems can be used more reliably for both proving theorems and finding counterexamples.

In [4], Bachmair, Ganzinger, and Waldmann introduced the hierarchical superposition calculus as a generalization of the superposition calculus for black-box style theory reasoning. Their calculus works in a framework of hierarchic specifications. It tries to prove the unsatisfiability of a set of clauses with respect to interpretations that extend a background model such as the integers with linear arithmetic conservatively, that is, without identifying distinct elements of old sorts ("confusion") and without adding new elements to old sorts ("junk"). While confusion can be detected by first-order theorem

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 39–57, 2013.

proving techniques, junk can not – in fact, the set of logical consequences of a hierarchic specifications is usually not recursively enumerable. Refutational completeness can therefore only be guaranteed if one restricts oneself to sets of formulas where junk can be excluded a priori. The property introduced by Bachmair, Ganzinger, and Waldmann for this purpose is called "sufficient completeness with respect to simple instances". Given this property, their calculus is refutationally complete for clause sets that are fully abstracted (i. e., where no literal contains both foreground and background symbols). Unfortunately their full abstraction rule may destroy sufficient completeness with respect to simple instances. We show that this problem can be avoided by using a new form of clause abstraction and a suitably modified hierarchical superposition calculus. Since the new hierarchical superposition calculus is still refutationally complete and the new abstraction rule is guaranteed to preserve sufficient completeness with respect to simple instances, the new combination is strictly more powerful than the old one.

In practice, sufficient completeness is a rather restrictive property. While there are application areas where one knows in advance that every input is sufficiently complete, in most cases this does not hold. As a user of an automated theorem prover, one would like to see a best effort behaviour: The prover might for instance try to *make* the input sufficiently complete by adding further theory axioms. In the calculus by Bachmair, Ganzinger, and Waldmann, however, this does not help at all: The restriction to a particular kind of instantiations ("simple instances") renders theory axioms essentially unusable in refutations. We show that this can be prevented by introducing two kinds of variables of the background theory sorts instead of one, that can be instantiated in different ways, making our calculus significantly "more complete" in practice. We also include a definition rule in the calculus that can be used to establish sufficient completeness by linking foreground terms to background parameters, thus allowing the background prover to reason about these terms.

The following trivial example demonstrates the problem. Consider the clause set $N = \{C\}$ where $C = \mathsf{f}(1) < \mathsf{f}(1)$. Assume that the background theory is integer arithmetic and that f is an integer-sorted operator from the foreground (free) signature. Intuitively, one would expect N to be unsatisfiable. However, N is not sufficiently complete, and it admits models in which $\mathsf{f}(1)$ is interpreted as some junk element $\not\in$, an element of the domain of the integer sort that is not a numeric constant. So both the calculus in [4] and ours are excused to not find a refutation. To fix that, one could add an instance $C' = \neg(\mathsf{f}(1) < \mathsf{f}(1))$ of the irreflexivity axiom $\neg(x < x)$. The resulting set $N' = \{C,\ C'\}$ is (trivially) sufficiently complete as it has no models at all. However, the calculus in [4] is not helped by adding C', since the abstracted version of N' is again not sufficiently complete and admits a model that interprets $\mathsf{f}(1)$ as $\not\in$. Our abstraction mechanism always preserves sufficient completeness and our calculus will find a refutation.

With this example one could think that replacing the abstraction mechanism in [4] with ours gives all the advantages of our calculus. This is not the case, however. Let $N'' = \{C,\ \neg(x < x)\}$ be obtained by adding the more realistic axiom $\neg(x < x)$. The set N'' is still sufficiently complete with our approach thanks to having two kinds of variables at disposal, but it is not sufficiently complete in the sense of [4]. Indeed, in that calculus adding background theory axioms *never* helps to gain sufficient completeness, as variables there have only one kind.

Another alternative to make N sufficiently complete is by adding a clause that forces $\mathsf{f}(1)$ to be equal to some background domain element. For instance, one can add a "definition" for $\mathsf{f}(1)$, that is, a clause $\mathsf{f}(1) \approx \alpha$, where α is a fresh symbolic constant belonging to the background signature (a "parameter"). The set $N''' = \{C, \mathsf{f}(1) \approx \alpha\}$ is sufficiently complete and it admits refutations with both calculi. The definition rule in our calculus mentioned above will generate this definition automatically. Moreover, the set N belongs to a syntactic fragment for which we can guarantee not only sufficient completeness (by means of the definition rule) but also refutational completeness.

We present the new calculus in detail and provide a general completeness result, modulo compactness of the background theory, and a specific completeness result for clause sets over ground background-sorted terms that does not require compactness. We also report on experiments with a prototypical implementation on the TPTP problem library. Complete proofs, which are omitted here for lack of space, can be found in [7].

Related Work. The relation with the predecessor calculus in [4] is discussed above and also further below. What we say there also applies to other developments rooted in that calculus, [1, e. g.]. The specialised version of hierarchic superposition in [13] will be discussed in Sect. 7 below. The resolution calculus in [12] has built-in inference rules for linear (rational) arithmetic, but is complete only under restrictions that effectively prevent quantification over rationals. Earlier work on integrating theory reasoning into model evolution [5,6] lacks the treatment of background-sorted foreground function symbols. The same applies to the sequent calculus in [17], which treats linear arithmetic with built-in rules for quantifier elimination. The instantiation method in [9] requires an answer-complete solver for the background theory to enumerate concrete solutions of background constraints, not just a decision procedure. All these approaches have in common that they integrate specialized reasoning for background theories into a general first-order reasoning method. A conceptually different approach consists in using first-order theorem provers as (semi-)decision procedures for specific theories in DPLL(T)(-like) architectures [14,2,8]. Notice that in this context the theorem provers do not need to reason modulo background theories themselves, and indeed they don't. The calculus and system in [14], for instance, integrates superposition and DPLL(T). From DPLL(T) it inherits splitting of ground non-unit clauses into their unit components, which determines a (backtrackable) model candidate M. The superposition inference rules are applied to elements from M and a current clause set F. The superposition component guarantees refutational completeness for pure first-order clause logic. Beyond that, for clauses containing background-sorted variables, (heuristic) instantiation is needed. Instantiation is done with ground terms that are provably equal w. r. t. the equations in M to some ground term in M in order to advance the derivation. The limits of that method can be illustrated with an (artificial but simple) example. Consider the unsatisfiable clause set $\{i \leq j \vee \mathsf{P}(i+1, x) \vee \mathsf{P}(j+2, x),\ i \leq j \vee \neg\mathsf{P}(i+3, x) \vee \neg\mathsf{P}(j+4, x)\}$ where i and j are integer-sorted variables and x is a foreground-sorted variable. Neither splitting into unit clauses, superposition calculus rules, nor instantiation applies, and so the derivation gets stuck with an inconclusive result. By contrast, the clause set belongs to a fragment that entails sufficient completeness ("no background-sorted foreground function symbols") and hence is refutable by our calculus. On the other hand, heuristic instantiation does have a place in our calculus, but we leave that for future work.

2 Signatures, Clauses, and Interpretations

We work in the context of standard many-sorted logic with first-order signatures comprised of sorts and operator (or function) symbols of given arities over these sorts. A *signature* is a pair $\Sigma = (\Xi, \Omega)$, where Ξ is a set of *sorts* and Ω is a set of *operator symbols* over Ξ. If $\mathcal{X}$ is a set of sorted variables with sorts in Ξ, then the set of well-sorted terms over $\Sigma = (\Xi, \Omega)$ and $\mathcal{X}$ is denoted by $\mathrm{T}_\Sigma(\mathcal{X})$; T_Σ is short for $\mathrm{T}_\Sigma(\emptyset)$. We require that Σ is a *sensible* signature, i. e., that T_Σ has no empty sorts. As usual, we write $t[u]$ to indicate that the term u is a (not necessarily proper) subterm of the term t. The position of u in t is left implicit.

A *Σ-equation* is an unordered pair (s, t), usually written $s \approx t$, where s and t are terms from $\mathrm{T}_\Sigma(\mathcal{X})$ of the same sort. For simplicity, we use equality as the only predicate in our language. Other predicates can always be encoded as a function into a set with one distinguished element, so that a non-equational atom is turned into an equation $P(t_1, \ldots, t_n) \approx true_p$; this is usually abbreviated by $P(t_1, \ldots, t_n)$.[1] A *literal* is an equation $s \approx t$ or a negated equation $\neg(s \approx t)$, also written as $s \not\approx t$. A *clause* is a multiset of literals, usually written as a disjunction; the empty clause, denoted by $\Box$ is a contradiction. If F is a term, equation, literal or clause, we denote by $vars(F)$ the set of variables that occur in F. We say F is *ground* if $\mathrm{vars}(F) = \emptyset$

A *substitution* σ is a mapping from variables to terms that is sort respecting, that is, maps each variable $x \in \mathcal{X}$ to a term of the same sort. Substitutions are homomorphically extended to terms as usual. We write substitution application in postfix form. A term s is an *instance* of a term t if there is a substitution σ such that $t\sigma = s$. All these notions carry over to equations, literals and clauses in the obvious way.

The *domain* of a substitution σ is the set $\mathrm{dom}(\sigma) = \{x \mid x \neq x\sigma\}$. We work with substitutions with finite domains only, written as $\sigma = [x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ where $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$. A *ground substitution* is a substitution that maps every variable in its domain to a ground term. A *ground instance* of F is obtained by applying some ground substitution with domain (at least) $\mathrm{vars}(F)$ to it.

A *Σ-interpretation* I consists of a Ξ-sorted family of carrier sets $\{I_\xi\}_{\xi \in \Xi}$ and of a function $I_f : I_{\xi_1} \times \cdots \times I_{\xi_n} \to I_{\xi_0}$ for every $f : \xi_1 \ldots \xi_n \to \xi_0$ in Ω. The *interpretation* t^I of a ground term t is defined recursively by $f(t_1, \ldots, t_n)^I = I_f(t_1^I, , \ldots, t_n^I)$ for $n \geq 0$. An interpretation I is called *term-generated*, if every element of an I_ξ is the interpretation of some ground term of sort ξ. An interpretation I is said to *satisfy* a ground equation $s \approx t$, if s and t have the same interpretation in I; it is said to *satisfy* a negated ground equation $s \not\approx t$, if s and t do not have the same interpretation in I. The interpretation I *satisfies* a ground clause C if at least one of the literals of C is satisfied by I. We also say that a ground clause C is *true in* I, if I satisfies C; and that C is *false in* I, otherwise. A term-generated interpretation I is said to *satisfy* a non-ground clause C if it satisfies all ground instances $C\sigma$; it is called a *model* of a set N of clauses, if it satisfies all clauses of N.[2] We abbreviate the fact that I is a model of N by $I \models N$; $I \models C$ is short for

[1] Without loss of generality we assume that there exists a distinct sort for every predicate.

[2] This restriction to term-generated interpretations as models is possible since we are only concerned with refutational theorem proving, i. e., with the derivation of a contradiction.

$I \models \{C\}$. We say that N *entails* N', and write $N \models N'$, if every model of N is a model of N'; $N \models C$ is short for $N \models \{C\}$.

If $\mathcal{J}$ is a class of Σ-interpretations, a Σ-clause or clause set is called $\mathcal{J}$*-satisfiable* if at least one $I \in \mathcal{J}$ satisfies the clause or clause set; otherwise it is called $\mathcal{J}$*-unsatisfiable*.

A *specification* is a pair $SP = (\Sigma, \mathcal{J})$, where Σ is a signature and $\mathcal{J}$ is a class of term-generated Σ-interpretations called *models* of the specification SP. We assume that $\mathcal{J}$ is closed under isomorphisms.

We say that a class of Σ-interpretations $\mathcal{J}$ or a specification $(\Sigma, \mathcal{J})$ is *compact*, if every infinite set of Σ-clauses that is $\mathcal{J}$-unsatisfiable has a finite subset that is also $\mathcal{J}$-unsatisfiable.

3 Hierarchic Theorem Proving

In hierarchic theorem proving, we consider a scenario in which a general-purpose foreground theorem prover and a specialized background prover cooperate to derive a contradiction from a set of clauses. In the sequel, we will usually abbreviate "foreground" and "background" by "FG" and "BG".

The BG prover accepts as input sets of clauses over a *BG signature* $\Sigma_\mathrm{B} = (\Xi_\mathrm{B}, \Omega_\mathrm{B})$. Elements of Ξ_B and Ω_B are called *BG sorts* and *BG operators*, respectively. We fix an infinite set $\mathcal{X}_\mathrm{B}$ of *BG variables* of sorts in Ξ_B. Every BG variable has (is labeled with) a *kind*, which is either *"abstraction"* or *"ordinary"*. Terms over Σ_B and $\mathcal{X}_\mathrm{B}$ are called *BG terms*. A BG term is called *pure*, if it does not contain ordinary variables; otherwise it is *impure*. These notions apply analogously to equations, literals, clauses, and clause sets.

The BG prover decides the satisfiability of Σ_B-clause sets with respect to a *BG specification* $(\Sigma_\mathrm{B}, \mathcal{B})$, where $\mathcal{B}$ is a class of term-generated Σ_B-interpretations called *BG models*. We assume that $\mathcal{B}$ is closed under isomorphisms.

In most applications of hierarchic theorem proving, the set of BG operators Ω_B contains a set of distinguished constant symbols $\Omega_\mathrm{B}^D \subseteq \Omega_\mathrm{B}$ that has the property that $d_1^I \neq d_2^I$ for any two distinct $d_1, d_2 \in \Omega_\mathrm{B}^D$ and every BG model $I \in \mathcal{B}$. We refer to these constant symbols as *(BG) domain elements*.

While we permit arbitrary classes of BG models, in practice the following three cases are most relevant:

(1) $\mathcal{B}$ consists of exactly one Σ_B-interpretation (up to isomorphism), say, the integer numbers over a signature containing all integer constants as domain elements and $\leq, <, +, -$ with the expected arities. In this case, $\mathcal{B}$ is trivially compact; in fact, a set N of Σ_B-clauses is $\mathcal{B}$-unsatisfiable if and only if some clause of N is $\mathcal{B}$-unsatisfiable.

(2) Σ_B is extended by an infinite number of *parameters*, that is, additional constant symbols. While all interpretations in $\mathcal{B}$ share the same carrier sets $\{I_\xi\}_{\xi \in \Xi_\mathrm{B}}$ and interpretations of non-parameter symbols, parameters may be interpreted freely by arbitrary elements of the appropriate I_ξ. The class $\mathcal{B}$ obtained in this way is in general not compact; for instance the infinite set of clauses $\{\, n \leq \beta \mid n \in \mathbb{N} \,\}$, where β is a parameter, is unsatisfiable in the integers, but every finite subset is satisfiable.

(3) Σ_B is again extended by parameters, however, $\mathcal{B}$ is now the class of all interpretations that satisfy some first-order theory, say, the first-order theory of linear integer

arithmetic.[3] Since $\mathcal{B}$ corresponds to a first-order theory, compactness is recovered. It should be noted, however, that $\mathcal{B}$ contains non-standard models, so that for instance the clause set $\{n \leq \beta \mid n \in \mathbb{N}\}$ is now satisfiable (e. g., $\mathbb{Q} \times \mathbb{Z}$ with a lexicographic ordering is a model).

The FG theorem prover accepts as inputs clauses over a signature $\Sigma = (\Xi, \Omega)$, where $\Xi_B \subseteq \Xi$ and $\Omega_B \subseteq \Omega$. The sorts in $\Xi_F = \Xi \setminus \Xi_B$ and the operator symbols in $\Omega_F = \Omega \setminus \Omega_B$ are called *FG sorts* and *FG operators*. Again we fix an infinite set $\mathcal{X}_F$ of *FG variables* of sorts in Ξ_F. All FG variables have the kind "ordinary". We define $\mathcal{X} = \mathcal{X}_B \cup \mathcal{X}_F$.

In examples we use $\{0, 1, 2, \ldots\}$ to denote BG domain elements, $\{+, -, <, \leq\}$ to denote (non-parameter) BG operators, and the possibly subscripted letters $\{x, y\}$, $\{X, Y\}$, $\{\alpha, \beta\}$, and $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{f}, \mathsf{g}\}$ to denote ordinary variables, abstraction variables, parameters, and FG operators, respectively. The letter ζ denotes an ordinary variable or an abstraction variable.

We call a term in $T_\Sigma(\mathcal{X})$ a *FG term*, if it is not a BG term, that is, if it contains at least one FG operator or FG variable (and analogously for equations, literals, or clauses). We emphasize that for a FG operator $\mathsf{f} : \xi_1 \ldots \xi_n \to \xi_0$ in Ω_F any of the ξ_i may be a BG sort, and that consequently FG terms may have BG sorts.

If I is a Σ-interpretation, the *restriction* of I to Σ_B, written $I|_{\Sigma_B}$, is the Σ_B-interpretation that is obtained from I by removing all carrier sets I_ξ for $\xi \in \Xi_F$ and all functions I_f for $f \in \Omega_F$. Note that $I|_{\Sigma_B}$ is not necessarily term-generated even if I is term-generated. In hierarchic theorem proving, we are only interested in Σ-interpretations that extend some model in $\mathcal{B}$ and neither collapse any of its sorts nor add new elements to them, that is, in Σ-interpretations I for which $I|_{\Sigma_B} \in \mathcal{B}$. We call such a Σ-interpretation a *$\mathcal{B}$-interpretation*.

Let N and N' be two sets of Σ-clauses. We say that N *entails* N' *relative to* $\mathcal{B}$ (and write $N \models_{\mathcal{B}} N'$), if every model of N whose restriction to Σ_B is in $\mathcal{B}$ is a model of N'. Note that $N \models_{\mathcal{B}} N'$ follows from $N \models N'$. If $N \models_{\mathcal{B}} \square$, we call N *$\mathcal{B}$-unsatisfiable*; otherwise, we call it *$\mathcal{B}$-satisfiable*.[4]

Our goal in refutational hierarchic theorem proving is to check whether a given set of Σ-clauses N is false in all $\mathcal{B}$-interpretations, or equivalently, whether N is $\mathcal{B}$-unsatisfiable.

We say that a substitution is *simple* if it maps every abstraction variable in its domain to a *pure* BG term. For example, $[x \mapsto 1 + Y + \alpha]$, $[X \mapsto 1 + Y + \alpha]$ and $[x \mapsto \mathsf{f}(1)]$ all are simple, whereas $[X \mapsto 1 + y + \alpha]$ and $[X \mapsto \mathsf{f}(1)]$ are not. Let F be a clause or (possibly infinite) clause set. By $sgi(F)$ we denote the set of simple ground instances of F, that is, the set of all ground instances of (all clauses in) F obtained by simple ground substitutions. Standard unification algorithms can be modified in a straightforward way for computing simple mgus. Note that a simple mgu can map an ordinary variable to an abstraction variable but not vice versa, as ordinary variables are not pure BG terms.

[3] To satisfy the technical requirement that all interpretations in $\mathcal{B}$ are term-generated, we assume that in this case Σ_B is suitably extended by an infinite set of constants (or by one constant and one unary function symbol) that are not used in any input formula or theory axiom.

[4] If $\Sigma = \Sigma_B$, this definition coincides with the definition of satisfiability w. r. t. a class of interpretations that was given in Sect. 2. A set N of BG clauses is $\mathcal{B}$-satisfiable if and only if some interpretation of $\mathcal{B}$ is a model of N.

For a BG specification $(\Sigma_B, \mathcal{B})$, we define $\mathrm{GndTh}(\mathcal{B})$ as the set of all ground Σ_B-formulas that are satisfied by every $I \in \mathcal{B}$.

Definition 3.1 (Sufficient completeness). *A Σ-clause set N is* sufficiently complete w. r. t. simple instances *iff for every Σ-model J of $sgi(N) \cup \mathrm{GndTh}(\mathcal{B})$[5] and every ground BG-sorted FG term s there is a ground BG term t such that $J \models s \approx t$.*[6]

For brevity, we will from now on omit the phrase "w. r. t. simple instances" and speak only of "sufficient completeness". It should be noted, though, that our definition differs from the classical definition of sufficient completeness in the literature on algebraic specifications.

4 Orderings

A *hierarchic reduction ordering* is a strict, well-founded ordering on terms that is compatible with contexts, i. e., $s \succ t$ implies $u[s] \succ u[t]$, and stable under simple substitutions, i. e., $s \succ t$ implies $s\sigma \succ t\sigma$ for every simple σ. In the rest of this paper we assume such a hierarchic reduction ordering $\succ$ that satisfies all of the following: (i) $\succ$ is total on ground terms, (ii) $s \succ d$ for every domain element d and every ground term s that is not a domain element, and (iii) $s \succ t$ for every ground FG term s and every ground BG term t. These conditions are easily satisfied by an LPO with an operator precedence in which FG operators are larger than BG operators and domain elements are minimal with, for example, $\cdots \succ -2 \succ 2 \succ -1 \succ 1 \succ 0$ to achieve well-foundedness.

Condition (iii) and stability under *simple* substitutions together justify to always order $s \succ X$ where s is a non-variable FG term and X is an abstraction variable. By contrast, $s \succ x$ can only hold if $x \in \mathrm{vars}(s)$. Intuitively, the combination of hierarchic reduction orderings and abstraction variables affords ordering more terms.

The ordering $\succ$ is extended to literals over terms by identifying a positive literal $s \approx t$ with the multiset $\{s, t\}$, a negative literal $s \not\approx t$ with $\{s, s, t, t\}$, and using the multiset extension of $\succ$. Clauses are compared by the multiset extension of $\succ$, also denoted by $\succ$.

The non-strict orderings $\succeq$ are defined as $s \succeq t$ iff $s \succ t$ or $s = t$ (the latter is multiset equality in case of literals and clauses). We say that a literal L is *maximal* (*strictly maximal*) in a clause $L \vee C$ iff there is no $K \in C$ with $K \succ L$ ($K \succeq L$).

5 Weak Abstraction

To refute an input set of Σ-clauses, hierarchic superposition calculi derive BG clauses from them and pass the latter to a BG prover. In order to do this, some separation of the FG and BG vocabulary in a clause is necessary. The technique used for this separation is known as *abstraction*: One (repeatedly) replaces some term t in a clause by a new variable and adds a disequations to the clause, so that $C[t]$ is converted

[5] In contrast to [4], we include $\mathrm{GndTh}(\mathcal{B})$ in the definition of sufficient completeness. (This is independent of the abstraction method used; it would also have been useful in [4].)

[6] Note that J need *not* be a $\mathcal{B}$-interpretation.

into the equivalent clause $\zeta \not\approx t \vee C[\zeta]$, where ζ is a new (abstraction or ordinary) variable.

The calculus by Bachmair, Ganzinger, and Waldmann [4] works on "fully abstracted" clauses: Background terms occuring below a FG operator or in an equation between a BG and a FG term or vice versa are abstracted out until one arrives at a clause in which no literal contains both FG and BG operator symbols.

A problematic aspect of any kind of abstraction is that it tends to increase the number of incomparable terms in a clause, which leads to an undesirable growth of the search space of a theorem prover. For instance, if we abstract out the subterms t and t' in a ground clause $\mathsf{f}(t) \approx \mathsf{g}(t')$, we get $x \not\approx t \vee y \not\approx t' \vee \mathsf{f}(x) \approx \mathsf{g}(y)$, and the two new terms $\mathsf{f}(x)$ and $\mathsf{g}(y)$ are incomparable in any reduction ordering. The approach used in [4] to reduce this problem is to consider only instances where BG-sorted variables are mapped to BG terms: In the terminology of the current paper, all BG-sorted variables in [4] have the kind "abstraction". This means that, in the example above, we obtain the two terms $\mathsf{f}(X)$ and $\mathsf{g}(Y)$. If we use an LPO with a precedence in which f is larger than g and g is larger than every BG operator, then for every simple ground substitution τ, $\mathsf{f}(X)\tau$ is strictly larger that $\mathsf{g}(Y)\tau$, so we can still consider $\mathsf{f}(X)$ as the only maximal term in the literal.

The advantage of full abstraction is that this clause structure is preserved by all inference rules. There is a serious drawback, however: Consider the clause set $N = \{\, 1 + \mathsf{c} \not\approx 1 + \mathsf{c} \,\}$. Since N is ground, we have $\mathrm{sgi}(N) = N$, and since $\mathrm{sgi}(N)$ is unsatisfiable, N is trivially sufficiently complete. Full abstraction turns N into $N' = \{\, X \not\approx \mathsf{c} \vee 1+X \not\approx 1+X \,\}$. In the simple ground instances of N', X is mapped to all pure BG terms. However, there are Σ-interpretations of $\mathrm{sgi}(N')$ in which c is interpreted differently from any pure BG term, so $\mathrm{sgi}(N') \cup \mathrm{GndTh}(\mathcal{B})$ does have a Σ-model and N' is no longer sufficiently complete. In other words, the calculus of [4] is refutationally complete for clause sets that are fully abstracted and sufficiently complete, but full abstraction may destroy sufficient completeness. (In fact, the calculus is not able to refute N'.)

The problem that we have seen is caused by the fact that full abstraction replaces FG terms by abstraction variables, which may not be mapped to FG terms later on. The obvious fix would be to use ordinary variables instead of abstraction variables whenever the term to be abstracted out is not a pure BG term, but as we have seen above, this would increase the number of incomparable terms and it would therefore be detrimental to the performance of the prover.

Full abstraction is a property that is stronger than actually necessary for the completeness proof of [4]. In fact, it was claimed in a footnote in [4] that the calculus could be optimized by abstracting out only non-variable BG terms that occur below a FG operator. This is incorrect, however: Using this abstraction rule, neither our calculus nor the calculus of [4] would not be able to refute $\{\, 1 + 1 \approx 2,\ (1 + 1) + \mathsf{c} \not\approx 2 + \mathsf{c} \,\}$, even though this set is unsatisfiable and trivially sufficiently complete. We need a slightly different abstraction rule to avoid this problem:

Definition 5.1. *A BG term t occurring in a clause C is called* target term *if t is neither a domain element nor a variable*[7] *and if C has the form $C[f(s_1, \ldots, t, \ldots, s_n)]$, where f is a FG operator or at least one of the s_i is a FG or impure BG term.*

A clause is called weakly abstracted *if it does not have any target terms.*

The weakly abstracted version of a clause *is the clause that is obtained by exhaustively replacing $C[t]$ by*

- *$C[X] \vee X \not\approx t$, where X is a new abstraction variable, if t is a pure target term in C,*
- *$C[y] \vee y \not\approx t$, where y is a new ordinary variable, if t is an impure target term in C.*

The weakly abstracted version of C is denoted by $\mathrm{abstr}(C)$.

For example, weak abstraction of the clause $\mathsf{g}(1, \alpha, \mathsf{f}(1)+(\alpha+1), z) \approx \beta$ yields $\mathsf{g}(1, X, \mathsf{f}(1)+Y, z) \approx \beta \vee X \not\approx \alpha \vee Y \not\approx \alpha + 1$. Note that the terms 1, $\mathsf{f}(1) + (\alpha + 1)$, z, and β are not abstracted out: 1 is a domain element; $\mathsf{f}(1) + (\alpha + 1)$ has a BG sort, but it is not a BG term; z is a variable; and β is not a subterm of a FG term. The clause $\mathsf{write}(\mathsf{a}, 2, \mathsf{read}(\mathsf{a}, 1)+1) \approx \mathsf{b}$ is already weakly abstracted. Every BG clause is trivially weakly abstracted.

Proposition 5.2. *If N is a set of clauses and N' is obtained from N by replacing one or more clauses by their weakly abstracted versions, then $\mathrm{sgi}(N)$ and $\mathrm{sgi}(N')$ are equivalent and N' is sufficiently complete whenever N is.*

In contrast to full abstraction, the weak abstraction rule does not require abstraction of FG terms (which can destroy sufficient completeness, if done using abstraction variables, and which is detrimental to the performance of a prover if done using ordinary variables). BG terms are usually abstracted out using abstraction variables. The exception are BG terms that are impure, i. e., that contain ordinary variables themselves. In this case, we cannot avoid to use ordinary variables for abstraction, otherwise, we might again destroy sufficient completeness. For example, the clause set $\{\, \mathsf{P}(1 + y),\ \neg\mathsf{P}(1 + \mathsf{c}) \,\}$ is sufficiently complete. If we used an abstraction variable instead of an ordinary variable to abstract out the impure subterm $1+y$, we would get $\{\, \mathsf{P}(X) \vee X \not\approx 1+y,\ \neg\mathsf{P}(1+\mathsf{c}) \,\}$, which is no longer sufficiently complete.

In input clauses (that is, before abstraction), BG-sorted variables may be declared as "ordinary" or "abstraction". As we have seen above, using abstraction variables can reduce the search space; on the other hand, abstraction variables may be detrimental to sufficient completeness. Consider the following example: The set of clauses $N = \{\, \neg\mathsf{f}(x) > \mathsf{g}(x) \vee \mathsf{h}(x) \approx 1, \neg\mathsf{f}(x) \leq \mathsf{g}(x) \vee \mathsf{h}(x) \approx 2, \neg\mathsf{h}(x) > 0 \,\}$ is unsatisfiable w. r. t. linear integer arithmetic, but since it is not sufficiently complete, the hierarchic superposition calculus does not detect the unsatisfiability. Adding the clause $X > Y \vee X \leq Y$ to N does not help: Since the abstraction variables X and Y may not be mapped to the FG terms $\mathsf{f}(x)$ and $\mathsf{g}(x)$ in a simple ground instance, the resulting set is still not sufficiently complete. However, if we add the clause $x > y \vee x \leq y$, the set of clauses becomes (vacuously) sufficiently complete and its unsatisfiability is detected.

One might wonder whether it is also possible to gain anything if the abstraction process is performed using ordinary variables instead of abstraction variables. The following proposition shows that this is not the case:

[7] The reason why it is permissible to treat domain elements in a special way will become clear in Sect. 6.

Proposition 5.3. *Let N be a set of clauses, let N' be the result of weak abstraction of N as defined above, and let N'' be the result of weak abstraction of N where all newly introduced variables are ordinary variables. Then sgi(N') and sgi(N'') are equivalent and sgi(N') is sufficiently complete if and only if sgi(N'') is.*

6 Base Inference System

An *inference system* $\mathcal{I}$ is a set of inference rules. By an $\mathcal{I}$ *inference* we mean an instance of an inference rule from $\mathcal{I}$ such that all conditions are satisfied.

The *base inference system* $\mathrm{HSP}_{\mathrm{Base}}$ of the hierarchic superposition calculus consists of the inference rules Equality resolution, Negative superposition, Positive superposition, Equality factoring, and Close defined below.[8] All inference rules are applicable only to weakly abstracted premise clauses.

$$\text{Equality resolution}\ \frac{s \not\approx t \vee C}{\mathrm{abstr}(C\sigma)}$$

if (i) neither s nor t is a pure BG term, (ii) σ is a simple mgu of s and t, and (iii) $(s \not\approx t)\sigma$ is maximal in $(s \not\approx t \vee C)\sigma$.[9]

For example, Equality resolution is applicable to $1 + \mathsf{c} \not\approx 1 + x$ with the simple mgu $[x \mapsto \mathsf{c}]$, but it is not applicable to $1 + \alpha \not\approx 1 + x$, since $1 + \alpha$ is a pure BG term.

$$\text{Negative superposition}\ \frac{l \approx r \vee C \qquad s[u] \not\approx t \vee D}{\mathrm{abstr}((s[r] \not\approx t \vee C \vee D)\sigma)}$$

if (i) neither l nor u is a pure BG term, (ii) u is not a variable, (iii) σ is a simple mgu of l and u, (iv) $r\sigma \not\succeq l\sigma$, (v) $(l \approx r)\sigma$ is strictly maximal in $(l \approx r \vee C)\sigma$, (vi) $t\sigma \not\succeq s\sigma$, and (vii) $(s \not\approx t)\sigma$ is maximal in $(s \not\approx t \vee D)\sigma$.

$$\text{Positive superposition}\ \frac{l \approx r \vee C \qquad s[u] \approx t \vee D}{\mathrm{abstr}((s[r] \approx t \vee C \vee D)\sigma)}$$

if (i) neither l nor u is a pure BG term, (ii) u is not a variable, (iii) σ is a simple mgu of l and u, (iv) $r\sigma \not\succeq l\sigma$, (v) $(l \approx r)\sigma$ is strictly maximal in $(l \approx r \vee C)\sigma$, (vi) $t\sigma \not\succeq s\sigma$, and (vii) $(s \not\approx t)\sigma$ is strictly maximal in $(s \approx t \vee D)\sigma$.

$$\text{Equality factoring}\ \frac{l \approx r \vee s \approx t \vee C}{\mathrm{abstr}((l \approx t \vee r \not\approx t \vee C)\sigma)}$$

where (i) neither l nor s is a pure BG term, (ii) σ is a simple mgu of l and s, (iii) $(l \approx r)\sigma$ is maximal in $(l \approx r \vee s \approx t \vee C)\sigma$, (iv) $r\sigma \not\succeq l\sigma$, and (v) $t\sigma \not\succeq s\sigma$.

$$\text{Close}\ \frac{C_1 \quad \cdots \quad C_n}{\square}$$

[8] With weak abstraction, it is *not* possible to replace Equality factoring by Factoring and Merging paramodulation. The inference system can be extended by selection functions, but only negative FG literals in clauses that do not contain ordinary BG variables may be selected.

[9] As in [4], it is possible to strengthen condition (iii) by requiring that there exists some simple ground substitution ψ such that $(s \not\approx t)\sigma\psi$ is maximal in $(s \not\approx t \vee C)\sigma\psi$ (and analogously for the other inference rules).

if $C_1, \ldots, C_n$ are BG clauses and $\{C_1, \ldots, C_n\}$ is $\mathcal{B}$-unsatisfiable, i. e., no interpretation in $\mathcal{B}$ is a Σ_{B}-model of $\{C_1, \ldots, C_n\}$.

Notice that Close is not restricted to take *pure* BG clauses only. The reason is that also impure BG clauses admit simple ground instances that are pure.

In contrast to [4], the inference rules above include an explicit weak abstraction in their conclusion. Without it, conclusions would not be weakly abstracted in general. For example Positive superposition applied to the weakly abstracted clauses $\mathsf{f}(X) \approx 1 \vee X \not\approx \alpha$ and $\mathsf{P}(\mathsf{f}(1)+1)$ would then yield $\mathsf{P}(1+1) \vee 1 \not\approx \alpha$, whose P-literal is not weakly abstracted. Additionally, the side conditions of our rules differ somewhat from the corresponding rules of [4], this is due on the one hand to the presence of impure BG terms (which must sometimes be treated like FG terms), and on the other hand to the fact that, after weak abstraction, literals may still contain both FG and BG operators.

The inference rules are supplemented by a redundancy criterion, that is, a mapping $\mathcal{R}_{\mathrm{Cl}}$ from sets of formulae to sets of formulae and a mapping $\mathcal{R}_{\mathrm{Inf}}$ from sets of formulae to sets of inferences that are meant to specify formulae that may be removed from N and inferences that need not be computed. ($\mathcal{R}_{\mathrm{Cl}}(N)$ need not be a subset of N and $\mathcal{R}_{\mathrm{Inf}}(N)$ will usually also contain inferences whose premises are not in N.)

Definition 6.1. *A pair $\mathcal{R} = (\mathcal{R}_{\mathrm{Inf}}, \mathcal{R}_{\mathrm{Cl}})$ is called a* redundancy criterion *(with respect to an inference system $\mathcal{I}$ and a consequence relation $\models$), if the following conditions are satisfied for all sets of formulae N and N':*

(i) $N \setminus \mathcal{R}_{\mathrm{Cl}}(N) \models \mathcal{R}_{\mathrm{Cl}}(N)$.
(ii) If $N \subseteq N'$, *then* $\mathcal{R}_{\mathrm{Cl}}(N) \subseteq \mathcal{R}_{\mathrm{Cl}}(N')$.
(iii) If ι *is an inference and its conclusion is in* N, *then* $\iota \in \mathcal{R}_{\mathrm{Inf}}(N)$.
(iv) If $N' \subseteq \mathcal{R}_{\mathrm{Cl}}(N)$, *then* $\mathcal{R}_{\mathrm{Inf}}(N) \subseteq \mathcal{R}_{\mathrm{Inf}}(N \setminus N')$.

Inferences in $\mathcal{R}_{\mathrm{Inf}}(N)$ and formulae in $\mathcal{R}_{\mathrm{Cl}}(N)$ are said to be redundant *with respect to N.*

To define a redundancy criterion for $\mathrm{HSP}_{\mathrm{Base}}$ and to prove the refutational completeness of the calculus, we use the same approach as in [4] and relate $\mathrm{HSP}_{\mathrm{Base}}$ inferences to the corresponding ground inferences in the standard superposition calculus SSP [16].

Let N be a set of ground clauses. We define $\mathcal{R}^{\mathcal{S}}_{\mathrm{Cl}}(N)$ to be the set of all clauses C such that there exist clauses $C_1, \ldots, C_n \in N$ that are smaller than C with respect to $\succ$ and $C_1, \ldots, C_n \models C$. We define $\mathcal{R}^{\mathcal{S}}_{\mathrm{Inf}}(N)$ to be the set of all ground SSP inferences ι such that either a premise of ι is in $\mathcal{R}^{\mathcal{S}}_{\mathrm{Cl}}(N)$ or else C_0 is the conclusion of ι and there exist clauses $C_1, \ldots, C_n \in N$ that are smaller with respect to $\succ^{\mathrm{c}}$ than the maximal premise of ι and $C_1, \ldots, C_n \models C_0$. It is well known that ground SSP together with $(\mathcal{R}^{\mathcal{S}}_{\mathrm{Inf}}, \mathcal{R}^{\mathcal{S}}_{\mathrm{Cl}})$ is refutationally complete.

Let ι be an $\mathrm{HSP}_{\mathrm{Base}}$ inference with premises $C_1, \ldots, C_n$ and conclusion abstr(C), where the clauses $C_1, \ldots, C_n$ have no variables in common. Let ι' be a ground SSP inference with premises $C'_1, \ldots, C'_n$ and conclusion C'. If σ is a simple substitution such that $C' = C\sigma$ and $C'_i = C_i\sigma$ for all i, and if none of the C'_i is a BG clause, then ι' is called a *simple ground instance* of ι. The set of all simple ground instances of an inference ι is denoted by sgi(ι).

Definition 6.2. *Let N be a set of weakly abstracted clauses. We define $\mathcal{R}^{\mathcal{H}}_{\mathrm{Inf}}(N)$ to be the set of all inferences ι such that either ι is not a Close inference and sgi(ι) $\subseteq \mathcal{R}^{\mathcal{S}}_{\mathrm{Inf}}(\mathrm{sgi}(N) \cup$*

$\mathrm{GndTh}(\mathcal{B})$), *or else* ι *is a Close inference and* $\Box \in N$. *We define* $\mathcal{R}^{\mathcal{H}}_{\mathrm{Cl}}(N)$ *to be the set of all weakly abstracted clauses* C *such that* $sgi(C) \subseteq \mathcal{R}^{\mathcal{S}}_{\mathrm{Cl}}(sgi(N) \cup \mathrm{GndTh}(\mathcal{B})) \cup \mathrm{GndTh}(\mathcal{B})$.[10]

To prove that $\mathrm{HSP}_{\mathrm{Base}}$ and $\mathcal{R}^{\mathcal{H}} = (\mathcal{R}^{\mathcal{H}}_{\mathrm{Inf}}, \mathcal{R}^{\mathcal{H}}_{\mathrm{Cl}})$ are refutationally complete for sets of weakly abstracted Σ-clauses and compact BG specifications $(\Sigma_{\mathrm{B}}, \mathcal{B})$, we use the same technique as in [4]:

First we show that $\mathcal{R}^{\mathcal{H}}$ is a redundancy criterion with respect to $\models_{\mathcal{B}}$, and that a set of clauses remains sufficiently complete if new clauses are added or if redundant clauses are deleted. The proofs for both properties are similar to the corresponding ones in [4]; the differences are due, on the one hand, to the fact that we include $\mathrm{GndTh}(\mathcal{B})$ in the redundancy criterion and in the definition of sufficient completeness, and, on the other hand, to the explicit abstraction steps in our inference rules.

We then encode arbitrary term-generated Σ_{B}-interpretation by sets of unit ground clauses in the following way: Let $I \in \mathcal{B}$ be a term-generated Σ_{B}-interpretation. For every Σ_{B}-ground term t let $m(t)$ be the smallest ground term of the congruence class of t in I. We define a rewrite system E'_I by $\mathrm{E}'_I = \{ t \to m(t) \mid t \in \mathrm{T}_\Sigma,\ t \neq m(t) \}$. Obviously, E'_I is terminating, right-reduced, and confluent. Now let E_I be the set of all rules $l \to r$ in E'_I such that l is not reducible by $\mathrm{E}'_I \setminus \{l \to r\}$. It is fairly easy to prove that E'_I and E_I define the same set of normal forms, and from this we can conclude that E_I and E'_I induce the same equality relation on ground Σ_{B}-terms. We identify E_I with the set of clauses $\{ t \approx t' \mid t \to t' \in \mathrm{E}_I \}$. Let D_I be the set of all clauses $t \not\approx t'$, such that t and t' are distinct ground Σ_{B}-terms in normal form with respect to E_I.

Let N be a set of weakly abstracted clauses and $I \in \mathcal{B}$ be a term-generated Σ_{B}-interpretation, then N_I denotes the set $\mathrm{E}_I \cup \mathrm{D}_I \cup \{ C\sigma \mid \sigma$ simple, reduced with respect to $\mathrm{E}_I, C \in N, C\sigma$ ground $\}$.

Theorem 6.3. *Let* $I \in \mathcal{B}$ *be a term-generated* Σ_{B}*-interpretation and let* N *be a set of weakly abstracted* Σ*-clauses. If* I *satisfies all BG clauses in* $sgi(N)$ *and* N *is saturated with respect to* $\mathrm{HSP}_{\mathrm{Base}}$ *and* $\mathcal{R}^{\mathcal{H}}$*, then* N_I *is saturated with respect to* SSP *and* $\mathcal{R}^{\mathcal{S}}$.

The crucial property of abstracted clauses that is needed in the proof of this theorem is that there are no superposition inferences between clauses in E_I and FG ground instances $C\sigma$, or in other words, that all FG terms occurring in ground instances $C\sigma$ are reduced w. r. t. E_I. Abstracting out FG terms as in [4] is not necessary to achieve this goal, and domain elements can also be excluded as target terms in Def. 5.1: Since two different domain elements must always be interpreted differently in I and since domain elements are smaller in the term ordering than any ground term that is not a domain element, every domain element is the smallest term in its congruence class. Domain elements occurring within FG terms are therefore trivially irreducible by E_I, so it is unnecessary to abstract them out.

If N is saturated with respect to $\mathrm{HSP}_{\mathrm{Base}}$ and $\mathcal{R}^{\mathcal{H}}$ and does not contain the empty clause, then Close cannot be applicable to N. If $(\Sigma_{\mathrm{B}}, \mathcal{B})$ is compact, this implies that there is some term-generated Σ_{B}-interpretation $I \in \mathcal{B}$ that satisfies all BG clauses in

[10] In contrast to [4], we include $\mathrm{GndTh}(\mathcal{B})$ in the redundancy criterion. (This is independent of the abstraction method used; it would also have been useful in [4].)

sgi(N). Hence, by Thm. 6.3, the set of *reduced simple* ground instances of N has a model that also satisfies $\mathrm{E}_I \cup \mathrm{D}_I$. Sufficient completeness allows us to show that this is in fact a model of *all* ground instances of clauses in N and that I is its restriction to Σ_B:

Theorem 6.4. *If the BG specification $(\Sigma_\mathrm{B}, \mathcal{B})$ is compact, then* $\mathrm{HSP}_\mathrm{Base}$ *and $\mathcal{R}^\mathcal{H}$ are refutationally complete for all sets of clauses that are sufficiently complete.*

We do not spell out in detail theorem proving *processes* here, because the well-known framework of standard resolution [3] can be readily instantiated with our calculus. In particular, it justifies the following version of a generic simplification rule for clause sets.

$$\textsf{Simp}\ \frac{N \cup \{C\}}{N \cup \{D\}}$$

if (i) D is weakly abstracted, (ii) $\mathrm{GndTh}(\mathcal{B}) \cup N \cup \{C\} \models D$, and (iii) C is redundant w. r. t. $N \cup \{D\}$.

Condition (ii) is needed for soundness, and condition (iii) is needed for completeness. The Simp rule covers the usual simplification rules of the standard superposition calculus, such as demodulation by unit clauses and deletion of tautologies and (properly) subsumed clauses. It also covers simplification of arithmetic terms, e. g., replacing a subterm $(2 + 3) + \alpha$ by $5 + \alpha$ and deleting an unsatisfiable BG literal $5 + \alpha < 4 + \alpha$ from a clause. Any clause of the form $C \vee \zeta \not\approx d$ where d is domain element can be simplified to $C[\zeta \mapsto d]$.

7 Sufficient Completeness by Define

In this section we introduce an additional inference rule, Define. It augments the $\mathrm{HSP}_\mathrm{Base}$ inference system with complementary functionality: while the $\mathrm{HSP}_\mathrm{Base}$ inference system will derive a contradiction if the input clause set is inconsistent and sufficiently complete, the Define rule may turn input clause sets that are not sufficiently complete into sufficiently complete ones. Technically, the Define rule derives "definitions" of the form $t \approx \alpha$, where t is a ground BG-sorted FG term and α is a parameter. This way, sufficient completenessis is achieved "locally" for t, by forcing t to be equal to some element of the carrier set of the proper sort, denoted by the parameter α. For economy of reasoning, definitions are introduced only on a by-need basis, when t appears in a current clause, and $t \approx \alpha$ is used to simplify that clause immediately.

We need one more preliminary definition before introducing Define formally.

Definition 7.1 (Unabstracted clause). *A clause is* unabstracted *if it does not contain any disequation $\zeta \not\approx t$ between a variable ζ and a term t unless $t \neq \zeta$ and $\zeta \in vars(t)$.*

Every clause can be unabstracted by repeatedly replacing $C \vee \zeta \not\approx t$ by $C[\zeta \mapsto t]$ whenever $t = \zeta$ or $\zeta \notin \mathrm{vars}(t)$. By unabstr($C$) we denote an unabstracted version of C that can be obtained this way.[11] If $t = t[\zeta_1, \ldots, \zeta_n]$ is a term in C and ζ_i is finally

[11] In general, unabstraction does not yield a unique result. All results are equivalent, however, and we can afford to select any one and disregard the others.

instantiated to t_i, we denote its unabstracted version $t[t_1, \ldots, t_n]$ by unabstr(t, C). For a clause set N let unabstr$(N) = \{\text{unabstr}(C) \mid C \in N\}$.

$$\textsf{Define}\ \frac{N \cup \{L[t[\zeta_1, \ldots, \zeta_n]] \vee D\}}{N \cup \{\text{abstr}(t[t_1, \ldots, t_n] \approx \alpha_{t[t_1,\ldots,t_n]}),\ \text{abstr}(L[\alpha_{t[t_1,\ldots,t_n]}] \vee D)\}}$$

if (i) $t[\zeta_1, \ldots, \zeta_n]$ is a minimal BG-sorted non-variable term with a toplevel FG operator, (ii) $t[t_1, \ldots, t_n] = \text{unabstr}(t[\zeta_1, \ldots, \zeta_n], L[t[\zeta_1, \ldots, \zeta_n]] \vee D)$, (iii) $t[t_1, \ldots, t_n]$ is ground, and (iv) $\alpha_{t[t_1,\ldots,t_n]}$ is a parameter, uniquely determined by the term $t[t_1, \ldots, t_n]$.

In condition (i), by minimality we mean that no proper subterm of $t[\zeta_1, \ldots, \zeta_n]$ is a BG-sorted non-variable term with a toplevel FG operator. In effect, the Define rule eliminates such terms inside-out. Conditions (iii) and (iv) are needed for soundness. Notice the Define-rule preserves $\mathcal{B}$-satisfiability, not $\mathcal{B}$-equivalence. In our main application, Thm. 7.5 below, every ζ_i will always be an abstraction variable.

Example 7.2. Consider the weakly abstracted clauses $\mathsf{P}(0)$, $\mathsf{f}(x) > 0 \vee \neg\mathsf{P}(x)$, $\mathsf{Q}(\mathsf{f}(x))$, $\neg\mathsf{Q}(x) \vee 0 > x$. Suppose $\neg\mathsf{P}(x)$ is maximal in the second clause. By superposition between the first two clauses we derive $\mathsf{f}(0) > 0$. With Define we obtain $\mathsf{f}(0) \approx \alpha_{\mathsf{f}(0)}$ and $\alpha_{\mathsf{f}(0)} > 0$, the latter replacing $\mathsf{f}(0) > 0$. From the third clause and $\mathsf{f}(0) \approx \alpha_{\mathsf{f}(0)}$ we obtain $\mathsf{Q}(\alpha_{\mathsf{f}(0)})$, and with the fourth clause $0 > \alpha_{\mathsf{f}(0)}$. Finally we apply Close to $\{\alpha_{\mathsf{f}(0)} > 0,\ 0 > \alpha_{\mathsf{f}(0)}\}$. □

In practice, it is interesting to identify conditions under which sufficient completeness can be established by means of Define *and* compactness poses no problems, so that a complete calculus results. The *ground BG-sorted term fragment (GBT fragment)* discussed below is one such case.

A clause set N belongs to the GBT fragment iff every clause $C \in N$ is a GBT clause, that is, all BG-sorted terms in C are ground. To get the desired completeness result we need to establish that the Define rule preserves the GBT property.

Lemma 7.3. *If* unabstr(N) *belongs to the GBT fragment and N' is obtained from N by a* Define *inference, then* unabstr(N') *also belongs to the GBT fragment.*

Below we will equip the HSP calculus with a specific strategy that first applies Define exhaustively before the derivation proper starts. In that, it may be beneficial to also apply Simp. But then, Simp needs to preserve the GBT property, too. Because this does not hold at the outset, and to make sure Split is well-behaved in the subsequent derivation, we have to make certain (mild) assumptions.

Definition 7.4. *Let $\succ_{fin}$ be any strict (partial) term ordering such that for every ground BG term s only finitely many ground BG terms t with $s \succ_{fin} t$ exist.*[12] *We say that a* Simp *inference with premise $N \cup \{C\}$ and conclusion $N \cup \{D\}$ is* suitable (for the GBT fragment) *iff (i) if* unabstr(C) *is a GBT clause then* unabstr(D) *is a GBT clause, (ii) for every BG term t occuring in* unabstr(D) *there is a BG term $s \in$* unabstr(C) *such that $s \succeq_{fin} t$, and (iii) every term t in D contains a BG-sorted FG operator only at toplevel position, if at all. We say the* Simp *inference rule is* suitable *iff every* Simp *inference is.*

[12] A KBO with appropriate weights can be used for $\succ_{\text{fin}}$.

Expected simplification techniques like demodulation, subsumption deletion and evaluation of BG subterms are all covered by suitable Simp rules. The latter is possible because simplifications are not only decreasing w. r. t. $\succ$ but *additionally* also decreasing w. r. t. $\succeq_{\text{fin}}$, as expressed in condition (ii). Without it, e. g., the clause $\mathsf{P}(1+1,0)$ would admit infinitely many simplified versions $\mathsf{P}(2,0)$, $\mathsf{P}(2,0+0)$, $\mathsf{P}(2,0+(0+0))$, etc. Condition (i) makes sure that also Simp preserves GBT clauses. Condition (iii) is needed to make sure that no new BG terms are generated in derivations.

As said, we need to equip the HSP calculus with a specific strategy. Assume a suitable Simp rule and let N be a set of GBT clauses. By N^{pre} we denote any clause set obtained by a derivation of the form $(N_0 = \text{abstr}(N)), N_1, \ldots, (N_k = N^{\text{pre}})$ with the inference rules Define and Simp only, and such that every $C \in N^{\text{pre}}$ either does not contain any BG-sorted FG operator or unabstr(C) is a ground positive unit clause of the form $\mathsf{f}(t_1,\ldots,t_n) \approx t$ where f is a BG-sorted FG operator and $t_1,\ldots,t_n,t$ do not contain BG-sorted FG operators.

For all GBT clause sets N, thanks to the effect of the Define rule and Lemma 7.3, all offending occurrences of BG-sorted FG terms in N can stepwisely be eliminated until a clause set N^{pre} results. Thanks to the additional assumptions about Simp we obtain the following main result on the GBT fragment.

Theorem 7.5. *The* HSP *calculus with a suitable* Simp *inference rule is refutationally complete for the ground BG-sorted term fragment. More precisely, if a set N of GBT clauses is $\mathcal{B}$-unsatisfiable then there is a refutation of N^{pre} without the* Define *rule.*

Because unabstraction can also be applied to fully abstracted clauses, it is possible to equip the hierarchic superposition calculus of [4] with a correspondingly modified Define rule and get Theorem 7.5 in that context as well.

In [13] it has been shown how to use hierarchic superposition as a decision procedure for ground clause sets (and for Horn clause sets with constants and variables as the only FG terms). Their method preprocesses the given clause set by "basification", a process that removes BG-sorted FG terms similarly as our Define rule. The resulting clause set then is fully abstracted and hierarchic superposition is applied. Certain modifications of the inference rules make sure derivations always terminate. Simplification is restricted to subsumption deletion. The effect of basification is achieved in our calculus by the Define rule. Moreover, for GBT clause sets, by Theorem 7.5, Define needs to be applied as preprocessing only. Applying Define beyond that for non-GBT clause sets can still be useful. Ex. 7.2, for instance, cannot be solved with basification during preprocessing. The fragment of [13] is a further restriction of the GBT fragment. We expect we can get decidability results for that fragment with similar techniques.

8 Implementation and Experiments

We have implemented the HSP calculus and carried out experiments with the TPTP Library [18]. Our implementation, "Beagle", is intended as a testbed for rapidly trying out theoretical ideas for their practical viability.[13] Beagle is in an early stage of

[13] `http://users.cecs.anu.edu.au/~baumgart/systems/beagle/`

development. Nevertheless it is a full implementation of HSP and accepts TPTP formulas over linear integer arithmetic ("TFF formulas", see [19]). The BG reasoner is a quantifier elimination procedure for linear integer arithmetic (LIA) based on Cooper's algorithm; it is called with all current BG clauses as inputs (with caching of simplified formulas) whenever a new BG clause has been derived. The HSP calculus itself is implemented in a straightforward way. Fairness is achieved through a combination of measuring clause lengths, depths and their derivation-age. Implemented simplification rules are evaluation of ground parameter-free BG terms and literals, expressing literals with the predicate symbols $\geq$ and $>$ in terms of $<$ and $\leq$, demodulation by unit clauses, proper subsumption deletion, and removing a positive literal L from a clause in presence of a unit clause that instantiates to the complement of L. Prover options allow the user to enable/disable the Define rule and to add certain LIA-valid clauses over ordinary variables. Unit clauses like $-(-x) \approx x$, $(x+(-y))+y \approx x$, $x+0 \approx x$, $x*0 \approx 0$, $\neg(x < x)$, etc, are always helpful as demodulators. Transitivity clauses for $<$ and $\leq$ are helpful sometimes. Optionally, a split rule can be enabled for branching out into complementary unit clauses if they simplify some current clause. Dependency-directed backtracking is used for search space pruning then. Beagle also implements the previous calculus [4], with abstraction variables only, full abstraction, and optionally a modified Define rule that uses full abstraction instead of weak abstraction. We refer to this setting by "HSPFA" below, and by "HSPWA" for the new calculus.

Beagle is implemented in Scala. The choice of a slow programming language and, more severely, the absence of any form of term indexing limit Beagle's applicability to small problems only. Indeed, Beagle's performance on problems that require significant combinatorial search is poor. For example, the propositional pigeonhole problem with 8 pigeons takes more than two hours, SPASS solves it in under 4 seconds using settings to get a comparable calculus and proof procedure (including splitting). Nevertheless we tried Beagle on all first-order problems from the TPTP library (version 5.4.0) over linear integer arithmetic. The experiments were run on a MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor. Here is our summary, by problem category.

ARI. Relevant are 223 problems. Many ARI problems are very simple, but roughly half of them are non-trivial by including integer-sorted non-constant FG function symbols and free predicates over the integers. The most difficult solved problems have a rating of 0.88. Beagle times out after 60 seconds on one problem (ARI184=1), terminates with an undecided result on one satisfiable problem (ARI603=1) and solves all other 221 problems correctly, 10 non-theorems and 211 theorems. All but three solved problems can be solved very quickly, the other three below 20 seconds. The Define rule is essential for HSPWA in 14 cases, for HSPFA in 17 cases. The problem ARI186=1 cannot be solved by HSPFA.

GEG. The five relevant problems are variations of each other. They deal with traversing weighted graphs and computing with the (sum of the) weights along paths. All five problems are solved within 60 seconds, the hardest problem has a rating of 0.67. For GEG025=1 the use of additional LIA-valid axioms, in particular transitivity of $\leq$ is essential. For two problems Define is essential, and one problem is unsolvable by HSPFA.

PUZ. The only relevant problem (PUZ133=2) is not solved.

NUM. The only non-easy problem that is solvable is NUM858=1 (rating 0.56), which is not solvable by HSPFA. All easy problems are solved easily, but for one problem Define is essential.

SEV/HWV. Six problems of SEV are relevant, about sets, stemming from a software verification context. Only SEV421=1 and SEV422=1 can be solved, in 3 and 100 seconds, respectively. The problem SEV422=1 cannot be solved by HSPFA. Solving the SEV-problems is dominated by pure foreground reasoning. The same applies to HWV, where no problem is solved.

SWV/SWW. Only one problem can be solved, SWV997=1 (rating 0.44), in 8 seconds. All other problems are too big to be converted into CNF in reasonable time. The same holds for all SWW problems.

SYO. Of the four problems, SYO521=1, SYO523=1 (rating 0.67) and SYO524=1 are solvable. The latter only with HSPWA, the Define rule and auxiliary lemmas.

The TSTP web page contains individual solutions to TPTP problems for various provers. About 12 provers are applicable to problems over linear integer arithmetic. Beagle solves 22 such problems with a rating of 0.60 or higher. Each of these problems can typically be solved by four or less systems, with CVC3, Princess, SPASS+T and Z3 the most reliable ones. There are five problems that only Princess and Beagle solve.

9 Conclusions

The main theoretical contribution of this paper is an improved variant of the hierarchic superposition calculus. The improvements over its predecessor [4] are grounded in a different form of "abstracted" clauses, the clauses the calculus works with internally. Because of that, a modified completeness proof is required. We have argued informally for the benefits over the old calculus in [4]. They concern making the calculus "more complete" in practice. It is hard to quantify that exactly in a general way, as completeness is impossible to achieve in presence of background-sorted foreground function symbols (e. g., "car" of integer-sorted lists). To compensate for that to some degree, we have reported on initial experiments with a prototypical implementation on the TPTP problem library. These experiments clearly indicate the benefits of our concepts, in particular the definition rule and the possibility of adding background theory axioms. They also confirm advantages of the new calculus over the old, the former solves strictly more more problems than the latter (and is never slower on the common set). Certainly more experimentation and an improved implementation is needed to also solve bigger-sized problems with a larger combinatorial search space.

We have also obtained a specific completeness result for clause sets over ground background-sorted terms and that does not require compactness. As far as we know this result is new. It is loosely related to the decidability results in [13], as discussed in Sect. 7. It is also loosely related to results in SMT-based theorem proving. For instance, the method in [11] deals with the case that variables appear only as arguments of, in our words, foreground operators. It works by ground-instantiating all variables in order to being able to use an SMT-solver for the quantifier-free fragment. Under certain conditions, finite ground instantiation is possible and the method is complete, otherwise it is

complete only modulo compactness of the background theory (as expected). Treating different fragments, the theoretical results are mutually non-subsuming with ours. Yet, on the fragment they consider we could adopt their technique of finite ground instantiation before applying Thm. 7.5 (when it applies). However, according to Thm. 7.5 our calculus needs instantiation of *background-sorted variables only*, this way keeping reasoning with foreground-sorted terms on the first-order level, as usual with superposition.

References

1. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition modulo linear arithmetic SUP(LA). In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 84–99. Springer, Heidelberg (2009)
2. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Trans. Comput. Log. 10(1) (2009)
3. Bachmair, L., Ganzinger, H.: Resolution Theorem Proving. In: Handbook of Automated Reasoning. North Holland (2001)
4. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational theorem proving for hierarchic first-order theories. Appl. Algebra Eng. Commun. Comput. 5, 193–212 (1994)
5. Baumgartner, P., Fuchs, A., Tinelli, C.: ME(LIA) – Model Evolution With Linear Integer Arithmetic Constraints. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 258–273. Springer, Heidelberg (2008)
6. Baumgartner, P., Tinelli, C.: Model evolution with equality modulo built-in theories. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 85–100. Springer, Heidelberg (2011)
7. Baumgartner, P., Waldmann, U.: Hierarchic superposition with weak abstraction. Technical Report MPI-I-2013-RG1-002, Max-Planck-Institut für Informatik, Saarbrücken, Germany (2013), `http://www.mpi-inf.mpg.de/publications/index.html`
8. Bonacina, M.P., Lynch, C., de Moura, L.M.: On deciding satisfiability by theorem proving with speculative inferences. J. Autom. Reasoning 47(2), 161–189 (2011)
9. Ganzinger, H., Korovin, K.: Theory instantiation. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 497–511. Springer, Heidelberg (2006)
10. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 167–182. Springer, Heidelberg (2007)
11. Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
12. Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007)
13. Kruglov, E., Weidenbach, C.: Superposition decides the first-order logic fragment over ground theories. Mathematics in Computer Science, 1–30 (2012)
14. de Moura, L.M., Bjørner, N.: Engineering DPLL(T) + saturation. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 475–490. Springer, Heidelberg (2008)
15. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)

16. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Handbook of Automated Reasoning, pp. 371–443. Elsevier and MIT Press (2001)
17. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008)
18. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
19. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP typed first-order form with arithmetic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 406–419. Springer, Heidelberg (2012)

Completeness and Decidability Results for First-Order Clauses with Indices*

Abdelkader Kersani and Nicolas Peltier

University of Grenoble (LIG, CNRS)

Abstract. We define a proof procedure that allows for a limited form of inductive reasoning. The first argument of a function symbol is allowed to belong to an inductive type. We will call such an argument an *index*. We enhance the standard superposition calculus with a loop detection rule, in order to encode a particular form of mathematical induction. The satisfiability problem is not semi-decidable, but some classes of clause sets are identified for which the proposed procedure is complete and/or terminating.

1 Introduction

We consider first-order clauses in which some of the function or predicate symbols are indexed by a particular kind of terms. The difference between these indices and the usual arguments is that the former are interpreted as ground terms constructed on a given signature (i.e. on an inductively defined domain), whereas the latter are interpreted arbitrarily (in the usual way). Consider the following example:

$$p_0(a) \wedge (\forall i\ \forall x\ p_i(x) \Rightarrow p_{s(i)}(f(x))) \wedge \forall x\ \neg p_n(x)$$

This formula is unsatisfiable if the constant n is interpreted as a term constructed on the signature $\{0, s\}$, i.e. as an element of $\mathbb{N}$. Indeed, for any $m \in \mathbb{N}$, the formula $p_m(f^m(a))$ can be derived from the first two conjuncts, yielding a contradiction with the third conjunct. However, if the indices are interpreted as ordinary terms, then the formula is obviously satisfiable. If the value of n is fixed (i.e. $n = 1, 2, \ldots$) then any first-order prover can easily establish the unsatisfiability of the formula. However, proving that it is unsatisfiable for every $n \in \mathbb{N}$ is a much harder problem, which obviously requires the use of mathematical induction. The previous formula can be viewed as a *schema* of clause sets, in the sense that, to transform this set into a standard clause set, one has to replace the "parameter" n by a ground term $s^m(0)$. Schemata of formulæ arise naturally in many applications of Automated Theorem Proving, in particular for formalizing parameterized systems (for instance circuits depending on the number of bits or layers [14]), for the modelling of dynamic systems (where the indices encode the time: $p_i(\boldsymbol{t})$ holds iff $p(\boldsymbol{t})$ is true at instant i), or for the formalization of inductive

* This work has been partly funded by the project ASAP of the French *Agence Nationale de la Recherche* (ANR-09-BLAN-0407-01).

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 58–75, 2013.

proofs in mathematics (the index then represents the induction parameter, see [5] for an example of use of this technique). In this paper, we devise a proof procedure for testing the satisfiability of such formulæ. The proposed inference system uses the usual rules of the superposition calculus (with a specific formalism in which parameters are abstracted away from the clauses), together with a new rule which encodes a form of mathematical induction. Due to well-known theoretical limitations, the satisfiability problem is not semi-decidable in general (see Proposition 1), but we devise some additional criteria that ensure completeness or termination. The indices are not necessarily natural numbers: they can be interpreted as any ground term on a given signature, provided all the (non-constant) symbols are monadic (i.e. the indices are interpreted as words).

The rest of the paper is structured as follows. In Section 2 we define the syntax and semantics of clausal logic with indices. In Section 3, we adapt the usual superposition calculus. In Section 4 we define a loop detection rule that strictly increases the power of the superposition calculus and we show examples of application. In Section 5 and 6, some abstract conditions ensuring refutational completeness and/or termination are devised. In Section 7 we provide an example of a syntactic class of clause sets fulfilling the previous conditions and Section 8 concludes the paper. Due to space restrictions, some proofs are omitted. Missing proofs can be found in [18].

1.1 Related Work

Our approach is strongly related to the "superposition calculus for fixed domain" procedure defined in [16]. Actually, the "superposition part" of our calculus is essentially equivalent to that in [16] and also to that in [7], which is designed to handle "hybrid" reasoning, i.e. reasoning combining the use of a theory-specific procedure with the superposition calculus for handling the generic part of the proof. However, in our approach the use of the fixed domain terms is more restricted: they only appear as distinguished arguments in the terms and *not* as ordinary arguments. Furthermore, we only consider formulæ with a unique parameter. This distinction between indices and ordinary terms reduces the scope of the method but permits to obtain much stronger completeness and decidability results. The proof procedure in [16] is not complete in general, since, for refuting the considered formula, an *infinite* set of empty constrained clauses must be generated in some cases. Some completeness and decidability results are presented in [15], however they are based on many additional conditions which do not hold in our case: all the clauses must be Horn, all the symbols must be monadic etc. The loop detection rule proposed in the present paper is also very different from the inductive rules defined in [16] and [15]. Therefore, the two approaches can be viewed as complementary (a more detailed comparison is provided in Section 4). Our work is also strongly related to inductive theorem proving. Explicit induction approaches (see for instance [10] or [8]) are often used by proof assistants, and powerful heuristics are employed to derive automatically the appropriate induction schemes [11]. Implicit induction schemes are used in rewrite-based theorem provers [9], whereas inductionless induction (see

for example [17,12]) uses proof by consistency to reduce the inductive validity to a mere satisfiability check. Very few completeness or termination results exist for such provers and our language does not fall in the scope of the known complete classes. In general, inductive theorem proving requires strong human guidance, especially for specifying the needed inductive lemmata. In contrast, our procedure, although more focused in this scope, is *purely automatic*: the loop detection rule allows one to generate automatically inductive invariants. Both the implicit and the inductionless induction approaches handle universal properties: the considered goals are of the form $\phi \models_{ind} \forall \boldsymbol{x} \psi$, where $\models_{ind}$ denotes the inductive logical consequence relation, ϕ is the axiomatization and ψ is a quantifier-free formula (usually a clause). Our work departs from these approaches because the goals we consider are rather of the form $\phi \models \forall n Q_1 x_1 \ldots Q_n x_n \psi$ (where n denotes the parameter, $x_1, \ldots, x_n$ are standard variables and $Q_1, \ldots, Q_n$ are quantifiers). Indeed the value of the standard first-order variables can possibly depend on the value of the parameter. Our calculus is also related to the proof procedure described in [1] which handles schemata of propositional formulæ indexed by integers. The scope of the present paper is however much larger, both for the base language (first-order logic instead of propositional logic) and for the type of the indices (terms – or words – instead of integers). Our calculus is also strictly more powerful than the one presented in [3], which only handles first-order clauses without equality. Besides, the completeness results in [3] only hold for purely propositional schemata.

2 Preliminaries

In this section, we define the syntax and semantics of the considered logic. We assume the reader is familiar with the usual notions in logic and automated deduction [22]. We consider first-order terms, built as usual on a sorted signature Σ and on a set of variables X, in which some of the (function or constant) symbols are indexed by terms of some special sorts. The sorts are used mainly to distinguish the indices from the ordinary terms. The set of sort symbols is thus partitioned into two disjoint sets $\mathcal{S}_I$ and $\mathcal{S}_T$, where $\mathcal{S}_I$ denotes the sorts of the indices and $\mathcal{S}_T$ the sorts of the ordinary terms. We assume that the profile of every non-constant symbol f is either of the form $\mathtt{s}_1 \times \ldots \times \mathtt{s}_n \rightarrow \mathtt{s}$, where $\mathtt{s}_2, \ldots, \mathtt{s}_n, \mathtt{s} \in \mathcal{S}_T$, and $\mathtt{s}_1 \in \mathcal{S}_I \cup \mathcal{S}_T$ or of the form $\mathtt{s} \rightarrow \mathtt{s}'$, where $\mathtt{s}, \mathtt{s}' \in \mathcal{S}_I$, i.e. all function symbols of a sort in $\mathcal{S}_T$ have at most one argument of a type in S_I and all (non-constant) function symbols of a sort in $\mathcal{S}_I$ are monadic and have a domain in $\mathcal{S}_I$. A *predicate symbol* is a function symbol of profile $\mathtt{s} \rightarrow \mathtt{bool}$. A variable of a sort in $\mathcal{S}_I$ is an *index variable*.

For readability, a term of head symbol $f : \mathtt{s}_1, \ldots, \mathtt{s}_n \rightarrow \mathtt{s}$ such that $\mathtt{s}_1 \in \mathcal{S}_I$ and $\mathtt{s}_2, \ldots, \mathtt{s}_n \in \mathcal{S}_T$ and of arguments $i, t_1, \ldots, t_n$ will be written $f_i(t_1, \ldots, t_n)$ (i is called an *index term*). The other terms are written as usual. This convention allows one to clearly distinguish the induction terms from the standard ones. If t is a term and p is a position (i.e. a finite sequence of natural numbers) then $t|_p$ denotes the subterm of t at position p (defined as usual). If v is a term then $t[v]_p$ denotes the term obtained from t by replacing the subterm at position p by v.

An *atom* is of the form $t_1 \simeq t_2$, where t_1, t_2 are terms of the same sort in $\mathcal{S}_T$ (equations between index terms are forbidden). A *literal* is either an atom (*positive* literal) or the negation of an atom (*negative* literal). A *clause* is a finite multiset of literals (written as a disjunction) and $\Box$ denotes the empty clause. Let α be a special constant symbol (not occurring in Σ) of a sort in $\mathcal{S}_I$. Throughout this paper, $\mathfrak{C}$ denotes some particular class of clauses.

Definition 1. *Let α be a constant symbol of a sort in S_I. An* α-clause *is an expression of the form $(\alpha \not\approx i_1) \vee \ldots \vee (\alpha \not\approx i_n) \vee C$ (with possibly $n = 0$) where:*

- *C is a clause.*
- *$i_1 \ldots i_n$ are terms of the same sort as α.*
- *Either C is empty, or all the variables in $i_1, \ldots, i_n$ occur in C.*

If, moreover, the clause C belongs to the class of clauses $\mathfrak{C}$, then $(\alpha \not\approx i_1) \vee \ldots \vee (\alpha \not\approx i_n) \vee C$ is an $(\alpha, \mathfrak{C})$-clause. *We call the α-clause* normalized *if $n \in \{0, 1\}$.*

The constant α is called the *parameter*. The class $\mathfrak{C}$ is mainly useful to restrict the syntactic form of the considered expressions. It is assumed to be fixed once and for all in the rest of the paper. Notice that, to avoid confusion, we use the symbol $\approx$ instead of $\simeq$ to denote equations between indices (i.e. in which α is involved). Notice also that the symbol α cannot occur in a term or literal (since $\alpha \notin \Sigma$). Therefore, a property such as $a_\alpha \simeq b$ for instance is to be written as $\alpha \not\approx x \vee a_x \simeq b$, where x is a variable. This idea of abstracting away terms and replacing them by variables is commonly used in the superposition framework to delay reasoning on some particular terms (see for instance [7]).

For every expression (term, atom, literal or clause) e, $\mathrm{var}(e)$ denotes the set of variables occurring in e. If $\mathrm{var}(e) = \emptyset$ then e is *ground*. A *substitution* σ is a function mapping every variable x to a term $x\sigma$ of the same sort as x. The *domain* $dom(\sigma)$ of σ is the set of variables x such that $x\sigma \neq x$. For every expression e, $e\sigma$ denotes the expression obtained from e by replacing every variable x by $x\sigma$. A substitution σ is *ground* iff for every $x \in dom(\sigma)$, $x\sigma$ is ground. A *renaming* is an injective substitution σ such that $x\sigma \in X$ for every $x \in dom(\sigma)$. The notions of *unifiers* and *most general unifiers* (*mgu*) are defined as usual. If t and s are two terms, we write $t \succeq s$ if $s = t\sigma$, for some substitution σ. We write $t \succ s$ if $t \succeq s$ and $s \not\succeq t$. A set of terms T of the same sort $\mathtt{s}$ is *covering for a term t* of sort $\mathtt{s}$ iff for every ground substitution θ, there exists $s \in T$ such that $t\theta \preceq s$. It is *covering* if it is covering for all terms of sort $\mathtt{s}$. The problem of testing whether a given set of terms is covering or not for a term t is decidable [13].

Definition 2. *An* interpretation $\mathcal{I}$ *is a pair $(=_\mathcal{I}, \mathcal{I}(\alpha))$ where $=_\mathcal{I}$ is a congruence on the ground terms whose sort is in $\mathcal{S}_T$ and $\mathcal{I}(\alpha)$ is a ground term of the same sort as α. An interpretation $\mathcal{I}$* validates*:*

- *A ground literal $t_1 \simeq t_2$ (resp. $t_1 \not\simeq t_2$) iff $t_1 =_\mathcal{I} t_2$ (resp. $t_1 \neq_\mathcal{I} t_2$).*
- *A ground clause $C \in \mathfrak{C}$ iff it validates at least one literal in C.*
- *A ground $(\alpha, \mathfrak{C})$-clause $\alpha \not\approx i_1 \cdots \vee \alpha \not\approx i_n \vee C$ iff either $\mathcal{I}(\alpha) \neq i_j$ for some $j \in [1, n]$ or $\mathcal{I}$ validates C.*

- *A non-ground* $(\alpha, \mathfrak{C})$*-clause* C *iff for every ground substitution* σ *of domain* $var(C)$, $\mathcal{I}$ *validates* $C\sigma$.
- *A set of* $(\alpha, \mathfrak{C})$*-clauses* S *iff it validates every* $(\alpha, \mathfrak{C})$*-clause in* S.

We write $\mathcal{I} \models S$ if $\mathcal{I}$ validates S ($\mathcal{I}$ is a *model* of S) and $S \models S'$ if every model of S is a model of S'.

The logic is obviously not decidable since it is clear that it encompasses first-order logic: if the clauses contain no occurrences of the special constant symbol α, then the value of the parameter is irrelevant, and an interpretation is simply a congruence on the set of Herbrand terms. In this case our semantics coincides with the usual one. The following theorem states that it is not even semi-decidable.

Theorem 1. *The unsatisfiability problem is not semi-decidable for* $(\alpha, \mathfrak{C})$*-clauses, if* $\mathfrak{C}$ *is the whole class of first-order clauses with one index variable.*

Proof. The proof is by reduction to the Post correspondence problem. Let $u^1, \ldots, u^n$ and $v^1, \ldots, v^n$ be two sequences of words. We construct a set of $(\alpha, \mathfrak{C})$-clauses S such that S is satisfiable iff there exists a sequence of indices $i_1, \ldots, i_m$ such that $u^{i_1}.\ldots.u^{i_m} = v^{i_1}.\ldots.v^{i_m}$. We use a constant symbol l encoding the sequence $i_1, \ldots, i_m$, with two function symbols *head* and *tail* returning respectively the head and the tail of a list. Words are encoded as usual, as lists of characters. The constant ϵ denotes the empty word and *concat* is a function concatenating two words (its definition is straightforward and is omitted). If $x \in \{u, v\}$ and y denotes a sequence of indices $i_1, \ldots, i_m$, then $sol(y, x)$ denotes the word $x^{i_1}.\ldots.x^{i_m}$. The terms $word(u, i)$ and $word(v, i)$ denote the words u^i and v^i respectively We use the following axioms:

- $\neg empty(x) \vee head(x) \simeq 1 \vee \ldots \vee head(x) \simeq n$ (if a sequence is not empty then its head is in $[1, n]$)
- $word(u, i) \simeq u^i$, for every $i \in [1, n]$ (definition of u).
- $word(v, i) \simeq v^i$, for every $i \in [1, n]$ (definition of v).
- $\neg empty(y) \vee sol(y, x) \simeq \epsilon$. If the sequence $i_1, \ldots, i_m$ is empty then the solution $x^{i_1}.\ldots.x^{i_m}$ is also empty.
- $empty(y) \vee sol(y, x) \simeq concat(word(x, head(y)), sol(tail(y), x))$. Otherwise, it corresponds to the concatenation of the word x^{i_1} and the solution corresponding to the sequence $i_2, \ldots, i_m$.
- $word(u, l) \simeq word(v, l)$. The two sequences are equal.

We now define a predicate $length_i(l)$ to encode the fact that l has length i.

- $length_0(x) \vee \neg empty(x)$
- $(\neg length_{i+1}(x) \vee \neg empty(x)) \wedge (\neg length_0(x) \vee empty(x))$
- $(length_{i+1}(x) \vee \neg length_i(tail(x))) \wedge (\neg length_{i+1}(x) \vee length_i(tail(x)))$

Finally, the following clauses state that the constant l denotes a finite list of length $\alpha \neq 0$ (notice that this property is the only one that cannot be expressed in first-order logic: otherwise l could denote an infinite or cyclic list): $\alpha \not\simeq x \vee length_x(l) \wedge \neg empty(l)$

It is straightforward to check that the previous set of clauses is satisfiable iff there exists a sequence of indices $i_1, \ldots, i_m$ satisfying the desired property.

Proposition 1. *Let $D : \alpha \not\approx i_1 \vee \ldots \vee \alpha \not\approx i_n \vee C$ be an $(\alpha, \mathfrak{C})$-clause, with $n \geq 1$. If $i_1, \ldots, i_n$ are not unifiable then D is a tautology. Otherwise, D is equivalent to $\alpha \not\approx i_1\sigma \vee C$ where σ is an mgu of $i_1, \ldots, i_n$. In this case, $\alpha \not\approx i_1\sigma \vee C$ is the* normalized form *of D.*

From now on, we assume that every $(\alpha, \mathfrak{C})$-clause is normalized. Indeed, by Proposition 1, a clause $\alpha \not\approx i_1 \vee \ldots \alpha \not\approx i_n \vee C$ can be either deleted or replaced by its normalized form.

Superposition calculus:

Superposition $C \vee t \simeq s, D \vee u \simeq v \rightarrow (C \vee D \vee t[v]_p \simeq s)\sigma$
if $\sigma = \mathrm{mgu}(u, t|_p)$, $u\sigma \not\leq v\sigma, t\sigma \not\leq s\sigma$, $t|_p$ is not a variable,
$(t \simeq s)\sigma \in \mathrm{sel}([C \vee t \simeq s]\sigma)$, $(u \simeq v)\sigma \in \mathrm{sel}([D \vee u \simeq v]\sigma)$.

Paramodulation $C \vee t \not\approx s, D \vee u \simeq v \rightarrow (C \vee D \vee t[v]_p \not\approx s)\sigma$
if $\sigma = \mathrm{mgu}(u, t|_p)$, $u\sigma \not\leq v\sigma, t\sigma \not\leq s\sigma$, $t|_p$ is not a variable,
$(t \not\approx s)\sigma \in \mathrm{sel}([C \vee t \not\approx s]\sigma)$, $(u \simeq v)\sigma \in \mathrm{sel}([D \vee u \simeq v]\sigma)$.

Reflection $C \vee t \not\approx s \rightarrow C\sigma$
if $\sigma = \mathrm{mgu}(t, s)$, $(t \not\approx s)\sigma \in \mathrm{sel}([C \vee t \not\approx s]\sigma)$.

Eq. Factorisation $C \vee t \simeq s \vee u \simeq v \rightarrow (C \vee s \not\approx v \vee t \simeq s)\sigma$
if $\sigma = \mathrm{mgu}(t, u)$, $t\sigma \not\leq s\sigma$, $u\sigma \not\leq v\sigma$, $(t \simeq s)\sigma \in \mathrm{sel}([C \vee t \simeq s \vee u \simeq v]\sigma)$.

Fig. 1. The superposition calculus

3 A Superposition Calculus for Indexed Clauses

Our proof procedure is a conservative extension of the superposition calculus [6,20]. All the rules are applied without any modification, except that disequations containing α are simply ignored (no inference can be applied from or into such disequations).

Let $<$ denote a simplification ordering that is total on ground terms [20]. The ordering $<$ is extended to atoms, literals and clauses using the multiset extension and to $(\alpha, \mathfrak{C})$-clauses simply by ignoring disequations containing α. A literal L is *maximal* in a clause $C \in \mathfrak{C}$ if for every $L' \in C$, $L \not< L'$. We consider a *selection function* sel which maps every clause C to a set of *selected literals* in C. For completeness, we assume that for every clause C, $\mathrm{sel}(C)$ contains either a negative literal or all maximal literals. This selection function is extended to $(\alpha, \mathfrak{C})$-clauses as follows: for every $(\alpha, \mathfrak{C})$-clause $\alpha \not\approx i_1 \vee \ldots \vee \alpha \not\approx i_n \vee C$, $\mathrm{sel}(\alpha \not\approx i_1 \vee \ldots \vee \alpha \not\approx i_n \vee C) \stackrel{\mathrm{def}}{=} \mathrm{sel}(C)$.

We consider the calculus (parameterized by $<$ and sel) of Figure 1. If S is a set of $(\alpha, \mathfrak{C})$-clauses, we write $S \vdash C$ if C is an $(\alpha, \mathfrak{C})$-clause and if there exists a non-tautological $(\alpha, \mathfrak{C})$-clause C' that can be deduced from S by applying one of the rules of Figure 1 such that C is the normalized form of C'.

We also adapt the usual redundancy criteria. A *tautology* is an $(\alpha, \mathfrak{C})$-clause containing two complementary literals, or a literal of the form $t \simeq t$, or a disjunction $\bigvee_{i=1}^{n} \alpha \not\approx t_i$, where $t_1, \ldots, t_n$ are not unifiable. An $(\alpha, \mathfrak{C})$-clause C is *subsumed* by an $(\alpha, \mathfrak{C})$-clause D if there exists a substitution σ such that $D\sigma \subseteq C$. A ground $(\alpha, \mathfrak{C})$-clause C is *redundant* in S if there exists a set of $(\alpha, \mathfrak{C})$-clauses S' such that $S' \models C$, and for every $D \in S'$, D is an instance of an $(\alpha, \mathfrak{C})$-clause in S such that $D \leq C$. A non ground $(\alpha, \mathfrak{C})$-clause C is *redundant* if all its instances are redundant. In particular, every subsumed $(\alpha, \mathfrak{C})$-clause and every tautological clause is redundant. A set of $(\alpha, \mathfrak{C})$-clauses S is *saturated* if every $(\alpha, \mathfrak{C})$-clause C such that $S \vdash C$ is redundant in S.

4 Loop Detection

The superposition calculus is not powerful enough to derive a contradiction from unsatisfiable sets of $(\alpha, \mathfrak{C})$-clauses, even in trivial cases, because it does not take into account the inductive structure of the domain of α. This is well illustrated by the following example.

Example 1. The first example in the Introduction can be encoded as the following set of $(\alpha, \mathfrak{C})$-clauses (where $p_i(t)$ denotes as usual the equation $p_i(t) \simeq \mathit{true}$).

$$1.\ p_0(a)\quad 2.\ \neg p_x(y) \vee p_{s(x)}(f(y))\quad 3.\ \alpha \not\approx x \vee \neg p_x(y)$$

It is easy to check that the following clauses can be generated by superposition:

4. $\alpha \not\approx 0$	(3,1)	7. $\alpha \not\approx s(s(x)) \vee \neg p_x(y)$	(5,2)
5. $\alpha \not\approx s(x) \vee \neg p_x(y)$	(3,2)	8. $\alpha \not\approx s(s(0))$	(7,1)
6. $\alpha \not\approx s(0)$	(5,1)	...	

It is clear that an infinite set of clauses of the form $\alpha \not\approx s^n(0)$ (for $n \in \mathbb{N}$) can be generated. Since α must be interpreted by a term of the form $s^n(0)$, the set is unsatisfiable (the set $\{s^n(0) \mid n \in \mathbb{N}\}$ is covering). However, no contradiction can be derived in finite time, which shows that the calculus is not complete (note that the logic is not compact).

In this section, we show how to overcome this problem in some cases (by Theorem 1 no general solution is possible). Notice that, in some cases, the calculus can generate a *finite* set of disequations $\{\alpha \not\approx t_1, \ldots, \alpha \not\approx t_n\}$ such that $\{t_1, \ldots, t_n\}$ is covering, in which case one may conclude that the clause set is unsatisfiable (of course it may also directly generate $\square$, if α is not involved in the proof). In order to handle the cases in which no such finite set of disequations can be generated, we define a loop detection rule, that encodes a form of mathematical induction (by "descente infinie") and that is able to derive clauses of the form $\alpha \not\approx t$ which cannot be derived by the superposition calculus. Intuitively, this rule applies when a cycle is detected in the search space, i.e. when S entails a set of $(\alpha, \mathfrak{C})$-clauses S' which is identical to S up to a shift of the value of the parameter α. The following definition formalizes this notion:

Definition 3. *A* shift *for a variable x is a substitution of the form $\{x \mapsto s\}$, where $s \neq x$ and $\mathit{var}(s) = \{x\}$. Let θ be a shift, C be a normalized $(\alpha, \mathfrak{C})$-clause and let t be a term with $\mathit{var}(t) = \{x\}$. The $(\alpha, \mathfrak{C})$-clause shift(C, t, θ) is defined as follows:*

- *If C is of the form $\alpha \not\approx t\sigma \vee D$, for some substitution σ and for some clause $D \in \mathfrak{C}$, then* $\mathit{shift}(C, t, \theta) \stackrel{def}{=} \alpha \not\approx t\theta\sigma \vee D$.
- *Otherwise* $\mathit{shift}(C, t, \theta) \stackrel{def}{=} C$.

If S is a set of $(\alpha, \mathfrak{C})$-clauses then $\mathit{shift}(S, t, \theta) \stackrel{def}{=} \bigcup_{C \in S} \{\mathit{shift}(C, t, \theta)\}$.

Example 2. Let $C : \alpha \not\approx f(g(x)) \vee h_{g(x)}(y) \simeq y$ and $D : \alpha \not\approx f(x) \vee h_x(y) \simeq a$. We have $\mathit{shift}(C, f(x), \{x \mapsto f'(x)\}) = \alpha \not\approx f(f'(g(x))) \vee h_{g(x)}(y) \simeq y$, $\mathit{shift}(C, f(g(x)), \{x \mapsto f'(x)\}) = \alpha \not\approx f(g(f'(x))) \vee h_{g(x)}(y) \simeq y$, $\mathit{shift}(D, f(x), \{x \mapsto f'(x)\}) = \alpha \not\approx f(f'(x)) \vee h_x(y) \simeq a$ and $\mathit{shift}(D, f(g(x)), \{x \mapsto f'(x)\}) = D$.

The loop detection rule is based on the following theorem. Intuitively, it applies when, for all possible instances s of some term t, the branch in the search space that corresponds to the case $\alpha = s$ is either closed (i.e. the clause set contains a clause of the form $\alpha \not\approx s'$, where $s' \succeq s$) or can be reduced (by shifting) to the branch corresponding to a strictly smaller term. Then the whole branch $\alpha = t$ can be closed, by "descente infinie".

A set of $(\alpha, \mathfrak{C})$-clauses S is a *t-set* if for every $(\alpha, \mathfrak{C})$-clause $\alpha \not\approx s \vee C$ occurring in S, we have $s \preceq t$.

Theorem 2. *Let S be a set of $(\alpha, \mathfrak{C})$-clauses. Assume there exists a set of terms $\{t_1, \ldots, t_n\}$ covering for t such that:*

1. *For all $i \in [1, n]$, there exists a t_i-set $S_i \subseteq S$.*
2. *For all $i \in [1, n]$ and for all ground terms $s \preceq t_i$, one of the following conditions holds:*
   (a) *$S_i \models \alpha \not\approx s$.*
   (b) *There exist a number $j \in [1, n]$ and a shift θ_s such that $S_i \models \mathit{shift}(S_j, t_j, \theta_s)$ and $s \preceq t_j\theta_s$.*

Then we have $S \models \{\alpha \not\approx t\}$.

Theorem 2 allows one to derive an $(\alpha, \mathfrak{C})$-clause of the form $\alpha \not\approx t$ from a set satisfying the previous properties. In practice, guessing the sets S_i and the shifts θ_s and checking whether (i) $S_i \models \alpha \not\approx s$ or (ii) $S_i \models \mathit{shift}(S_j, t_j, \theta_s)$ is of course infeasible. We need to impose stronger syntactic conditions. A simple solution (used in the sequel) is to check that: (i) a clause of the form $\alpha \not\approx s'$ with $s' \succeq s$ has been derived from parent clauses in S_i, (ii) $\mathit{shift}(S_j, t_i, \theta_s)$ has been derived from S_i. Of course, to apply the theorem in practice, one has to exhibit a *finite* set of substitutions θ_s that covers all possible terms, so that Condition 2 holds. The completeness proof of the next section provides additional hints on how the theorem should be applied in practice (in particular, to choose the sets S_i). From now on, we write $S \vdash C$ if an $(\alpha, \mathfrak{C})$-clause C can be deduced from a set of clause sets S by superposition or by the loop detection rule.

Example 3. Consider the set of $(\alpha, \mathfrak{C})$-clauses of Example 1. Let $t_1 = x$, $S_1 = \{1, 2, 3\}$ and $S' = \{1, 2, 5\}$. We check that the conditions of Theorem 2 are fulfilled. The set $\{t_1\}$ is obviously covering, and it is clear from the derivation in Example 1 that we have

$S_1 \vdash^* \alpha \not\approx 0$ and $S_1 \vdash^* S'$. Furthermore, it is easy to check that $S' = \mathit{shift}(S_1, x, \{x \mapsto s(x)\})$. Let s be a ground term. The term s is either 0, in which case Condition 2.a is satisfied, or of the form $s(t')$ for some term t', in which case Condition 2.b holds (with $j = 1$ and $\theta_s = \{x \mapsto s(x)\}$). Thus Theorem 2 applies and the clause $\alpha \not\approx x$ is generated. This clause is obviously unsatisfiable, which proves that the original clause set is also unsatisfiable.

If the indices are natural numbers, the conditions are much simpler to test, since all covering sets contain a subset of the form $\{0, s(0), \ldots, s^k(0), s^{k+1}(x)\}$. Thus we may assume that $k = 1$ and that there exists at most one shift θ_s.

The loop detection rule is related to the inductive rule presented in [16]. Both rules apply globally, on the whole clause set (and not on a fixed finite set of premises, as the usual inference rules). They both encode a form of mathematical induction in the context of a superposition-based calculus, with the aim of deriving a contradiction in some cases where the other rules diverge. However, there exist important differences between these two rules. The inductive rule of [16] encodes the fact that, when proving a formula $\phi(\alpha)$, where α ranges over some inductively defined domain, one may assume that α is minimal, i.e. that $\neg\phi(\beta)$ holds for every β strictly lower than α (according to some well-founded ordering). The purpose of the inductive rule is precisely to derive the formula $\neg\phi(\beta)$, with additional constraints ensuring that $\beta < \alpha$. Obviously, this rule only preserves satisfiability. In contrast, our rule uses induction in the form of descente infinie: it only applies when a formula $\phi(\beta)$ has been *explicitly* derived from $\phi(\alpha)$ by the inference rules. The only properties that can be derived are clauses of the form $\alpha \not\approx t$, which in some sense close the branches corresponding to instances of t in the search space. The conclusion is a logical consequence of the premises, and the inference strongly depends on the considered signature.

Example 4. For instance, consider the clause set $\{\alpha \not\approx x \vee p_x, \neg p_a, \neg p_{f(x)} \vee q_x, \neg q_{f(x)} \vee q_x, \neg q_a\}$. The superposition calculus derives the clauses: $\alpha \not\approx a, \alpha \not\approx f^n(f(x)) \vee q_x$ and $\alpha \not\approx f^n(f(a))$, for every $n \in \mathbb{N}$. The inductive rule of [16] applies on the initial clause set and derives (for instance) the clause: $\alpha \not\approx f(x) \vee \neg p_x$. This is intuitively justified by the fact that if we assume that α is the minimal term such that $\alpha \not\approx x \vee p_x$ (i.e. p_α), holds then necessarily p_x cannot hold if x is a proper subterm of α. Notice that this clause does not help to derive a contradiction in this case. Our rule cannot derive such a property. However, since the clauses $\alpha \not\approx f(f(x)) \vee \neg p_x$ and $\alpha \not\approx f(a)$ can be derived from $\alpha \not\approx f(x) \vee p_x$ (using the clauses not containing α), and since $\alpha \not\approx f(f(x)) \vee \neg p_x = \mathit{shift}(\alpha \not\approx f(x) \vee p_x, f(x), \{x \mapsto f(x)\})$, the loop detection rule applies and derives $\alpha \not\approx f(x)$. Notice that this is only possible because the signature only contains f and a (otherwise the set $\{f(x), a\}$ would not be covering). Together with the clause $\alpha \not\approx a$, this proves the unsatisfiability of the initial clause set.

The loop detection rule also departs from the technique presented in [15] for deciding the validity of $\forall\exists$-queries. The latter approach is based, roughly speaking, on a "compilation" of the search space into an automaton, and to a reduction of the satisfiability problem to the emptiness problem for the represented language.

We show a more complex example of application of our approach.

Example 5. Let T be an array. Let $a, b_1, \ldots, b_n$ be indices of T, such that $\forall i\ a \neq b_i$. We consider the array T' obtained from T by changing successively the value of the cell b_i to some constant c_i. We want to prove that $T[a] = T'[a]$. This is formalized in our setting by the following set of clauses. We define a sequence of arrays T_i with the following clauses:

$$(1)\quad T_0 \simeq T \qquad (2)\quad T_{i+1} \simeq \mathit{store}(T_i, b_i, c_i)$$

We have the axiom:

$$(3)\quad b_i \not\simeq a$$

We also consider the usual axioms of the theory of arrays (see for instance [4]):

$$(4)\quad \mathit{select}(\mathit{store}(t, x, v), x) \simeq v \qquad (5)\quad x \simeq y \vee \mathit{select}(\mathit{store}(t, x, v), y) \simeq \mathit{select}(t, y)$$

The inequation $T_n[a] \neq T[a]$ is defined as follows:

$$(6)\quad \mathit{select}(T, a) \simeq d \qquad (7)\quad \alpha \not\approx i \vee \mathit{select}(T_i, a) \not\simeq d$$

We then derive the following $(\alpha, \mathfrak{C})$-clauses by applying the superposition calculus:

$$\begin{array}{ll}
(8)\ \alpha \not\approx 0 \vee \mathit{select}(T, a) \not\simeq d & (1,7)\\
(9)\ \alpha \not\approx 0 & (6,8)\\
(10)\ y \approx b_i \vee \mathit{select}(T_{i+1}, y) \simeq \mathit{select}(T_i, y) & (2,5)\\
(11)\ \alpha \not\approx i+1 \vee a \simeq b_i \vee \mathit{select}(T_i, a) \not\simeq d & (10,7)\\
(12)\ \alpha \not\approx i+1 \vee a \simeq b_i \vee \mathit{select}(T, a) \not\simeq d & (1,11)\\
(13)\ \alpha \not\approx 1 \vee a \simeq b_0 & (6,12)\\
(14)\ \alpha \not\approx 1 & (13,3)\\
(15)\ \alpha \not\approx i+2 \vee a \simeq b_i \vee \mathit{select}(T_i, a) \not\simeq d & (10,11)
\end{array}$$

We check that the conditions of Theorem 2 hold. Consider the sets $S_1 = \{1, 6, 10, 11\}$ and $S' = \{1, 6, 10, 15\}$. S_1 is an $i+1$-set and S' is an $i+2$-set. We have $S_1 \vdash^* S'$. Furthermore, $S_1 = \mathit{shift}(S', i+1, \{i \mapsto i+1\})$. The only ground term that is an instance of $i+1$ but not of $i+2$ is 1, and we have $S_1 \vdash^* \alpha \not\approx 1$. Hence the looping rule applies, yielding (16) $\alpha \not\approx i+1$. Since $\{0, i+1\}$ is covering, Clauses 9 and 16 entail that the original clause set is unsatisfiable. Notice that the loop detection rule can be applied in several different ways. In this case, if we consider the sets $S_1 = \{1, 6, 10, 11\}$, $S_2 = \{9\}$ and $S' = \{1, 6, 10, 15\}$, then one can *directly* generate $\alpha \not\approx i$ (since S_2 is a 0-set, $S_2 \vdash^* \alpha \not\simeq 0$ and $\{0, 1, i+2\}$ is covering).

5 Completeness

Since the considered logic is not semi-decidable (by Theorem 1), the proof procedure cannot be complete in general. That is why we impose some additional conditions on the considered clause sets. At this point, we only introduce abstract, semantic conditions which are meant to be as general as possible and sufficient to ensure completeness. In Section 7 we will provide an example of a concrete (syntactic) class of clause sets fulfilling these requirements.

For technical convenience, we assume that the considered terms contain no constant symbols of a sort in $\mathcal{S}_I$. This condition greatly simplifies the definitions

and proofs. It is not really restrictive, since any constant symbol a may be replaced by a term $a(x)$, where x is a dummy variable[1]. Furthermore, we assume that for every $(\alpha, \mathfrak{C})$-clause $\alpha \not\approx t \vee C$ occurring in the considered clause sets, t is not a variable. Again this condition is not restrictive: it can be enforced by introducing a new function symbol f of profile $s \to s'$, where s is the initial sort of α and s' is a new sort symbol, and by replacing every disequation $\alpha \not\approx t$ (where t is possibly a variable) by $\alpha \not\approx f(t)$ (note that by definition f is the unique function symbol of sort s', thus for all interpretations $\mathcal{I}$, $\mathcal{I}(\alpha)$ will be of the form $f(\ldots)$). A clause C is *index-flat* if every index occurring in C is a variable.

Definition 4. *We denote by succ the partial function such that* $succ(t) \stackrel{def}{=} f_1(\ldots(f_{n-1}(x)))$ *if* t *is of the form* $f_1(\ldots(f_n(x))\ldots))$ *for some variable* x *(succ is undefined otherwise).*

We now introduce a function "rank" that plays a central role in the following. It maps all $(\alpha, \mathfrak{C})$-clauses C to the (necessarily unique) term t such that $\alpha \not\approx t$ occurs in C (or $\bot$ if α does not occur in C). However, if C is not index-flat, then we will return not the term t itself, but rather its successor according to *succ*. The reason behind this seemingly non-intuitive definition is that we want the rank to be preserved by instantiation (for the clauses whose indices have depth 0 or 1), for instance $\alpha \not\approx f(g(x)) \vee p_{g(x)}$ should have the same rank as $\alpha \not\approx f(x) \vee p_x$. More formally:

Definition 5. *The* rank *of an* $(\alpha, \mathfrak{C})$*-clause* C *is defined as follows.*

- *If* $C \in \mathfrak{C}$ *then* $rank(C) \stackrel{def}{=} \bot$.
- *If* C *is of the form* $\alpha \not\approx i \vee D$ *and* D *is index-flat then* $rank(C) \stackrel{def}{=} i$.
- *If* C *is of the form* $\alpha \not\approx i \vee D$ *and* D *is not index-flat then* $rank(C) \stackrel{def}{=} succ(i)$.

The function rank is well-defined, since i cannot be a variable. If S is a set of $(\alpha, \mathfrak{C})$-clauses, we denote by $S\langle r\rangle$ the set of $(\alpha, \mathfrak{C})$-clauses of rank r (up to a renaming) in S. We then consider a particular subset of $\mathfrak{C}$, obtained by considering only the index-flat $(\alpha, \mathfrak{C})$-clauses that can interact with a non-index-flat $(\alpha, \mathfrak{C})$-clause:

Definition 6. *We denote by* $\mathfrak{F}$ *the set of index-flat clauses* $C \in \mathfrak{C}$ *such that there exist two clauses* $D, E \in \mathfrak{C}$ *such that* $C, D \vdash E$ *and* D *is not index-flat.*

If S is a set of $(\alpha, \mathfrak{C})$-clauses, we denote by $\mathfrak{F}(S)$ the set $\{\alpha \not\approx t \vee C \in S \mid C \in \mathfrak{F}\}$.

Definition 7. *The class* $\mathfrak{C}$ *is* admissible *iff it satisfies the following conditions:*

(c_1*)* $\mathfrak{C}$ *is closed under superposition i.e. if* $S \vdash C$ *and* $S \subseteq \mathfrak{C}$ *then* $C \in \mathfrak{C}$.

[1] Of course the signature Σ must contain a constant symbol of the same sort as x, otherwise the set of ground terms would be empty. However, this constant is not allowed to occur in the clauses.

(c_2) *Each non-empty clause in* $\mathfrak{C}$ *contains exactly one index variable. Furthermore, if* C *and* D *contain two index variables* x *and* y *respectively, and if an inference is applicable on* C, D *with a unifier* σ*, then* $\sigma(x)$ *is* x*,* y *or of the form* $f(y)$*, for some function symbol* f*. Moreover if* $\sigma(x) = f(y)$ *then* C *is index-flat and* D *is not. Similarly, if* C *contains an index variable* x*, and if a unary inference is applicable on* C *with a unifier* σ*, then we must have* $\sigma(x) = x$*.*

(c_3) $\mathfrak{F}$ *is finite (up to a renaming of variables).*

From now on we assume that the considered class $\mathfrak{C}$ is admissible. Condition (c_1) is rather natural: it guarantees that the $(\alpha, \mathfrak{C})$-clauses constructed over $\mathfrak{C}$ are closed under superposition inferences. Condition (c_2) will ensure that these inferences cannot increase the depth of the indices arbitrarily. Condition (c_3) ensures that only finitely many clauses of a given rank can be built on $\mathfrak{F}$. As we will show, (c_2) entails that the search space has a strict hierarchic structure w.r.t. the rank: for all terms $t \succeq s$, the $(\alpha, \mathfrak{C})$-clauses of rank s are necessarily derived from those of rank t (along with the clauses of rank $\bot$). Furthermore, we will prove that this relation still holds if the clauses are restricted to those occurring in $\mathfrak{F}$, i.e. we have $t \succeq s \Rightarrow S\langle\bot\rangle \cup \mathfrak{F}(S\langle t\rangle) \vdash^* S\langle\bot\rangle \cup \mathfrak{F}(S\langle s\rangle)$ (assuming t is lower than the ranks of the initial $(\alpha, \mathfrak{C})$-clauses). Then, (c_3) ensures that there exist only finitely many sets $\mathfrak{F}(S\langle t\rangle)$ (up to a shift), which entails that the loop detection rule eventually applies. A clause set S is *saturated* if every clause C such that $S \vdash C$ is redundant w.r.t. S (in the usual sense). The following theorem states our completeness result:

Theorem 3. *Let* S *be a set of* $(\alpha, \mathfrak{C})$*-clauses where* $\mathfrak{C}$ *is admissible. If* S *is saturated and unsatisfiable then either* $\square$ *or* $\alpha \not\approx x$ *occurs in* S*.*

6 Satisfiability Detection

In standard clausal logic, the satisfiability of a given clause set S can sometimes be established by saturation, in case the set of clauses derived from S is finite and does not contain $\square$. This is not feasible in the context of this paper (except in trivial cases), because the rank will in general increase indefinitely. However, we can devise the following satisfiability test, based on a form of *partial* saturation:

Definition 8. *A set of* $(\alpha, \mathfrak{C})$*-clauses* S *is* saturated w.r.t. a (ground) term t *if for every clause* C *such that* $S \vdash C$*, either* C *is redundant or* C *is of the form* $\alpha \not\approx s \vee D$*, where* s *and* t *are not unifiable.*

If a set of $(\alpha, \mathfrak{C})$-clauses S is saturated w.r.t. some term t and does not contain $\alpha \not\approx t$, then it can be shown that S has a model in which the value of α is t (note that testing whether a clause set is saturated w.r.t. t is easy: if the standard proof search algorithm is used, it suffices to test that the set of "active" clauses contains no clause of a rank more general than t).

If $\mathfrak{C}$ is finite (i.e. if the superposition calculus terminates on clause sets in $\mathfrak{C}$), then the number of $(\alpha, \mathfrak{C})$-clauses of a fixed rank is also finite. This entails

that, for every ground term t, it is possible to eventually obtain, from any initial clause set S, a clause set containing S and saturated w.r.t. t. If, moreover, there exists a term t such that this partially saturated set does not contain $\alpha \not\approx t$, then satisfiability can be detected. If no such term exists, the initial clause set must be unsatisfiable, thus termination can be ensured in both cases, although the set of $(\alpha, \mathfrak{C})$-clauses is infinite.

Theorem 4. *If $\mathfrak{C}$ is finite, then the satisfiability problem is decidable for sets of $(\alpha, \mathfrak{C})$-clauses.*

Note that the fact that $\mathfrak{C}$ is finite does not imply that the set of terms that can be constructed on the signature is finite, since the restrictions on the superposition inferences can prevent such terms from being generated.

7 Complete Classes of Indexed Formulæ

In this section, we demonstrate the applicability of our results by providing an example of an admissible *syntactic* class of $(\alpha, \mathfrak{C})$-clauses.

Let μ be a *complexity function* mapping every ground term to a natural number. We assume that for any $k \in \mathbb{N}$, the set of ground objects t such that $\mu(t) \leq k$ and such that every index in t is of depth at most 1 is finite. Examples of usual complexity functions satisfying this requirement include the depth (maximal length of the non-index positions occurring in the terms) or the size (number of non-index positions in the terms). The function μ is extended to atom, literals and clauses as follows:

- $\mu(t \not\simeq s) \stackrel{\text{def}}{=} \mu(t \simeq s) \stackrel{\text{def}}{=} \max(\mu(t), \mu(s))$, and
- $\mu(\bigvee_{i=1}^{n} l_i) \stackrel{\text{def}}{=} \max\{\mu(l_i) \mid i \in [1, n]\}$.

We write $t =_\mu s$ if for every substitution σ we have $\mu(t\sigma) = \mu(s\sigma)$. For every natural number ν, we write $t \leq_\mu^\nu s$ if for every substitution σ such that $\mu(t\sigma) > \nu$, we have $\mu(t\sigma) \leq \mu(s\sigma)$. The relations $=_\mu$ and $\leq_\mu^\nu$ are hard to test in general because the set of substitutions σ is infinite. However, algorithms for testing whether $t =_\mu s$ or $t \geq_\mu^\nu s$ for various complexity functions μ are defined in [19,21]. For instance, if μ is the depth of the term, then it is easy to see that $t =_\mu s$ iff the following conditions hold: $\mu(t) = \mu(s)$, t and s have the same set of variables and the maximal occurrence depth of every variable is the same in t and in s.

We consider two (not necessarily disjoint) sets of predicate symbols: a set of *control predicates* Ω_c and a set of *index propagation predicates* Ω_i. A literal is a *control literal* (resp. an *index propagation literal*) if its atom is of the form $p_i(t_1, \ldots, t_n) \simeq \mathtt{true}$, where $p \in \Omega_c$ (resp. $p \in \Omega_i$) and i is an index term. The remaining literals are called the *standard literals*. Let $\mathcal{S}_p$ be a set of sort symbols, called the *μ-preserving sorts* satisfying the following properties:

- For every predicate symbol $p \in \Omega_c \cup \Omega_i$ of profile $\mathtt{s}_1 \times \ldots \times \mathtt{s}_n \to \mathtt{bool}$, and for every $i \in [1, n]$, we have $\mathtt{s}_i \in \mathcal{S}_p$.
- For every function symbol of profile $\mathtt{s}_1 \times \ldots \times \mathtt{s}_n \to \mathtt{s}$, if $\mathtt{s} \in \mathcal{S}_p$ then $\forall i \in [1, n], \mathtt{s}_i \in \mathcal{S}_p$.

Intuitively, the sorts in $\mathcal{S}_p$ are the sorts of the terms occurring in control or index propagation literals.

Definition 9. *A set of clauses S is* μ-controlled *iff the following conditions holds:*

1. *For every equation $t \simeq s$ occurring in a clause in S, if t and s are of a sort in $\mathcal{S}_p$, then $t =_\mu s$.*
2. *There exists a natural number ν such that, for every clause C, and for every index propagation literal or positive control literal L in C, we have $L \leq^\nu_\mu C'$, where C' is the set of negative control literals occurring in C.*
3. *All literals, except index propagation literals, are index-flat.*
4. *Every clause in S contains at most one index variable.*
5. *The atoms in S are either of the form $f_i(\boldsymbol{t}) \simeq g_i(\boldsymbol{s})$ (equational atoms) or of the form $p_i(\boldsymbol{t}) \simeq \mathtt{true}$ (non-equational atoms).*

In particular, any non-equational index-flat clause set is μ-controlled (with $\mathcal{S}_p = \Omega_c = \Omega_i = \emptyset$). Condition 1 ensures that the superposition inferences will not affect the complexity of the terms occurring inside control or index propagation literals. Condition 2 states that the complexity of every positive control literal and of every index propagation literals is dominated by the complexity of the negative control literals occurring in the clause. Condition 3 ensures that the only dependencies between terms of distinct indices are encoded by index-propagation literals (the remaining literals state relations involving only terms with the same indices). Condition 5 forbids equations between indexed and non-index non-boolean terms, such as $a_i \simeq b$. Equations between variables are also forbidden.

Example 6. Let μ be the depth function. Let $\Omega_c = \{p\}$, $\Omega_i = \{q\}$. The clauses $p_i(f(c))$, $\neg p_i(x) \vee p_{s(i)}(x) \vee \neg q_i(x)$, $p_a(b)$, $\alpha \not\simeq i \vee p_i(f(b))$, $q_i(x) \vee \neg q_i(f(x))$, $q_i(f(c))$, $p_i(x) \vee \neg r_i(y) \vee r_i(f(y))$, $r_i(a)$, are μ-controlled, with $\nu = 2$. Note that the conditions on the control literals ensure that for all literals of the form $p_i(t)$ or $q_i(t)$ generated by the inferences, t is of depth at most 2. In contrast, the literals of head r_i can be of arbitrary depth. The clauses $f(x) \simeq g(f(x))$, $\neg q_i(x) \vee q_i(f(x))$, $p_i(f(x)) \vee \neg q_i(x)$, $p_i(x)$, $a_{s(i)} \simeq b_i$, $p_i \vee p_j$, $f(x) \simeq g_i(x)$ are not μ-controlled, because they contradict Conditions $1, 2, 2, 2, 3, 4$ and 5, respectively. In particular, $f(x) \simeq g(f(x))$ contradicts Condition 1 because $f(x)$ and $g(f(x))$ have distinct depths. Similarly, $\neg q_i(x) \vee q_i(f(x))$ violates Condition 2 (regardless of the value of ν), because the depth of $q_i(x)$ is not asymptotically greater than that of $q_i(f(x))$.

The superposition strategy is defined as follows.

- Negative control literals are selected in any clause containing only control literal and index propagation literals. Otherwise, the selected literals are the maximal ones.
- The ordering satisfies the following properties:
  - The relation $p_{f(i)}(t_1, \ldots, t_n) > q_i(s_1, \ldots, s_m)$ holds for all symbols $p, q, t_1, \ldots, t_n, s_1, \ldots, s_m$ and f.
  - All standard atoms of index i are strictly greater than all index propagation atoms with the same index i.

The second condition on the ordering may seem rather strong, since, clearly, the considered index propagation atom can contain variables not occurring in the standard atom. However, it can easily be enforced by assuming that the terms occurring at a root level in the standard atoms are of some special sorts that cannot occur inside an index propagation atom (those terms can then be assumed to be strictly greater than the ones occurring in index propagation atoms).

Theorem 5. *The class of μ-controlled clauses is admissible.*

Proof. We prove simultaneously that Conditions c_1, c_2 and c_3 holds. In particular, c_3 is established by showing that every clause in $\mathfrak{F}$ contains no variable except index variables and is of complexity lower than ν (this clearly entails that $\mathfrak{F}$ is finite up to a renaming). We consider two μ-controlled clauses $C[t]_p$ and $u \simeq v \vee D$ and a clause $C[v]_p\sigma \vee D\sigma$, obtained by superposition from the two first clauses (the proof for the other inference rules is similar).

- Assume that $C[t]_p$ is index-flat and that $u \simeq v \vee D$ is not. Due to the ordering used to restrict the inferences, u cannot be index-flat (otherwise u would not be maximal). Therefore, u is of the form $p_{f(i)}(\boldsymbol{t})$, where $p \in \Omega_i$. Consequently, t is of the form $p_j(\boldsymbol{s})$ (since t and u are unifiable, they must have the same head symbol). But then since t is selected, $C[t]_p$ cannot contain any standard atom (otherwise t would not be maximal) neither any negative control literal (otherwise this literal would be selected). Thus the set of negative control literals in C is empty, and by Condition 2 in Definition 9, we have $\mu(C[t]_p) \leq^{\nu}_{\mu} \Box$. This implies that $C[t]_p$ contains no variables (except index variables) and is of complexity at most ν. The same reasoning holds if $u \simeq v \vee D$ is index-flat and $C[t]_p$ is not (in this case $u \simeq v \vee D$ contains no non-index variable and we must have $\mu(u \simeq v \vee D) \leq \nu$). Thus the clauses in $\mathfrak{F}$ contain no variable (except index variables) and are of complexity at most ν.
- Since the depth of the index terms is at most 1 and since, due to the ordering restrictions, only the literals with deepest indices are eligible for inferences, the condition c_2 is easy to check. Notice that this implies that the number of index variables does not increase.
- It only remains to prove that c_1 holds, i.e. that $C[v]_p\sigma \vee D\sigma$ is μ-controlled. We prove that this clause satisfies all the conditions of Definition 9.
  - Let $t \simeq s$ be an equation in $C[v]_p\sigma \vee D\sigma$, where t, s are of a sort in $\mathcal{S}_p$. If $t \simeq s$ is of the form $(t' \simeq s')\sigma$ where $t' \simeq s'$ occurs in one of the parent clauses, then since the parent clauses are μ-controlled by hypothesis, we have necessarily $t' =_\mu s'$ and thus also $t =_\mu s$. Otherwise, $t \simeq s$ is of the form $(t'[v]_q \simeq s')\sigma$, where $t' \simeq s'$ occurs in one of the parent clauses and $t'|_q = u$. Again, we have $t' =_\mu s'$. Furthermore, by definition of $\mathcal{S}_p$, u and v must be of a sort in $\mathcal{S}_p$. Consequently, we have also $u =_\mu v$ and thus $t' =_\mu t'[v]_q$. Therefore, $t'[v]_q\sigma =_\mu s'\sigma$.
  - Let L be a literal occurring in $C[v]_p\sigma \vee D\sigma$, that is either an index propagation literal or a positive control literal. By definition, L is obtained

from a literal L' occurring in one of the parent clauses by applying the substitution σ and by (possibly) replacing an occurrence of the term $u\sigma$ by $v\sigma$. If u does not occur in L then obviously $L = L'\sigma$, thus $L =_\mu L'\sigma$. If u occurs in L then by definition of $\mathcal{S}_p$, u must be of a sort in $\mathcal{S}_p$, thus we have $u =_\mu v$ and therefore in both cases the relation $L =_\mu L'\sigma$ holds. Since the parent clauses containing L' is μ-controlled, it also contains a disjunction of control literals M such that $M \geq^\nu_\mu L'$. Thus $C[v]_p\sigma \vee D\sigma$ contains a disjunction of literals M' obtained from $M\sigma$ by replacing $u\sigma$ by $v\sigma$. If $u \simeq v$ is a control literal, then D contains a disjunction of negative control literals D' such that $D' \geq^\nu_\mu u \simeq v$. Thus we have $D'\sigma \geq^\nu_\mu M\sigma \geq^\nu_\mu L$. Otherwise, if u occurs in M then it must be of a sort in $\mathcal{S}_p$. Therefore we have $u =_\mu v$, which implies that $M' =_\mu M\sigma$, and thus $M' \geq^\nu_\mu L$.

- Assume that $C[v]_p\sigma \vee D\sigma$ contains a literal L that is not an index propagation literal and that is not index-flat. This literal is obtained from a literal L' occurring in one of the parent clauses by applying the substitution σ and by replacing the term $u\sigma$ by $v\sigma$. L' cannot be an index propagation literal . Thus L' is index-flat, which means that σ cannot be flat and that (by Condition c_2) the parent clause not containing the literal L' must be non-index-flat. But we have shown that the only index-flat clauses that can interact with non-index-flat clauses only contain index propagation literals, which contradicts our initial assumption on L.
- Condition 4 is an immediate consequence of c_2.
- The last condition is straightforward to check, since it holds for the two parents and it is obviously preserved by replacement and instantiation.

Intuitively, the index propagation literals encode properties of the index terms and relations between them, whereas the other literals encode properties of standard terms (for a given index). The use of control literals ensures that the size of the former literals is bounded whereas the latter can be arbitrary first-order literals. Several concrete classes can be obtained simply by instantiating the complexity measure μ. All these classes are strictly more expressive than first-order logic (which corresponds to the case in which both Ω_c and Ω_i are empty). Theorem 3 ensures that our calculus is complete for μ-controlled clause sets. If, moreover, the considered clauses belong to a class for which the superposition calculus terminates (such as the monadic class, the guarded fragment,... or if all the literals are index propagation or control literals) then Theorem 4 ensures termination and decidability. We thus obtain – without any additional effort – a general completeness result for schemata of first-order clauses and decidability results for schemata built on decidable subclasses of first-order logic. In particular, it is easy to check (see [3]) that every regular propositional schema [1] can be expressed as a set of μ-controlled clauses (in this case all literals are index propagation literals and indices are natural numbers). Therefore, the formulæ of propositional linear temporal logic can also be encoded as μ-controlled clause sets (see [2] for a translation of LTL into regular schemata). Many properties of usual inductively defined data-structures such as lists or trees can be encoded

into controlled clauses. For instance, a tree can be denoted as a constant symbol τ indexed by positions[2], where τ_p denotes the label of the node at position p. Then we can easily encode the fact that some first-order property is satisfied by all labels in the tree (or by all labels occurring along some position, or some regular set of positions). However, we *cannot* express the fact that, e.g., a tree is sorted, because it requires to use atoms of the form $\tau_{p.0} \leq \tau_p \leq \tau_{p.1}$, which necessarily involve terms with several *distinct* indices. Relations between distinct trees can also be expressed, provided they preserve the shape of the tree (for instance we can state that a tree is obtained from τ by applying some function f on every node).

8 Conclusion

A proof procedure for handling clauses with indexed terms has been presented, enriching the superposition calculus with a carefully controlled form of inductive reasoning. Although the satisfiability problem is not even semi-decidable in general, criteria have been devised to ensure refutational completeness and termination. At the best of our knowledge, no other proof procedure provides similar features. Future work includes the implementation of this procedure and the evaluation of its practical performance. To this purpose, developing efficient algorithms to apply the loop detection rule is obviously essential (Sections 4 and 5 provide some hints in this direction). A first implementation has already been completed in the particular case in which the indices are natural numbers (defined on the signature $\{0, succ\}$) and will soon be available. From a more theoretical point of view, other classes of clause sets satisfying the conditions of Section 5 have to be identified. In particular, it would be interesting to find a terminating class containing the example provided in Section 4 concerning the theory of arrays (as well as examples from other similar theories: lists, records, integers etc.). Another line of future work is to extend the proof procedure in order to allow equations between indices.

References

1. Aravantinos, V., Caferra, R., Peltier, N.: Decidability and undecidability results for propositional schemata. Journal of Artificial Intelligence Research 40, 599–656 (2011)
2. Aravantinos, V., Caferra, R., Peltier, N.: Linear Temporal Logic and Propositional Schemata, Back and Forth. In: TIME 2011 (18th International Symposium on Temporal Representation and Reasoning) (2011)
3. Aravantinos, V., Echenim, M., Peltier, N.: A resolution calculus for first-order schemata. In: Fundamenta Informaticae (to appear 2013) (accepted for publication)
4. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Information and Computation 183(2), 140–164 (2003)

[2] It is clear that positions can be encoded in a monadic signature.

5. Baaz, M., Hetzl, S., Leitsch, A., Richter, C., Spohr, H.: CERES: An analysis of Fürstenberg's proof of the infinity of primes. Theor. Comput. Sci. 403(2-3), 160–175 (2008)
6. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. Journal of Logic and Computation 3(4), 217–247 (1994)
7. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational theorem proving for hierachic first-order theories. Appl. Algebra Eng. Commun. Comput. 5, 193–212 (1994)
8. Baelde, D., Miller, D., Snow, Z.: Focused inductive theorem proving. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 278–292. Springer, Heidelberg (2010)
9. Bouhoula, A., Kounalis, E., Rusinowitch, M.: SPIKE, an automatic theorem prover. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 460–462. Springer, Heidelberg (1992)
10. Bundy, A.: The automation of proof by mathematical induction. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 845–911. Elsevier and MIT Press (2001)
11. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: meta-level guidance for mathematical reasoning. Cambridge University Press, New York (2005)
12. Comon, H.: Inductionless induction. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 14, pp. 913–962. North-Holland (2001)
13. Comon, H., Lescanne, P.: Equational problems and disunification. Journal of Symbolic Computation 7, 371–475 (1989)
14. Gupta, A., Fisher, A.L.: Parametric circuit representation using inductive boolean functions. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 15–28. Springer, Heidelberg (1993)
15. Horbach, M., Weidenbach, C.: Deciding the inductive validity of $\forall \exists^*$ queries. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 332–347. Springer, Heidelberg (2009)
16. Horbach, M., Weidenbach, C.: Superposition for fixed domains. ACM Trans. Comput. Logic 11(4), 1–35 (2010)
17. Kapur, D., Musser, D.: Proof by consistency. Artificial Intelligence 31 (1987)
18. Kersani, A., Peltier, N.: Completeness and Decidability Results for First-order Clauses with Indices (long version), Research Report (2013), `http://membres-lig.imag.fr/peltier/kp13.pdf`
19. Leitsch, A.: Deciding clause classes by semantic clash resolution. Fundamenta Informaticae 18, 163–182 (1993)
20. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 371–443. Elsevier and MIT Press (2001)
21. Peltier, N.: Some Techniques for Proving Termination of the Hyperresolution Calculus. Journal of Automated Reasoning 35, 391–427 (2006)
22. Robinson, A., Voronkov, A. (eds.): Handbook of Automated Reasoning. North-Holland (2001)

A Proof Procedure for Hybrid Logic with Binders, Transitivity and Relation Hierarchies

Marta Cialdea Mayer

Università di Roma Tre, Italy

Abstract. A tableau calculus constituting a decision procedure for hybrid logic with the converse modalities, the global ones and a restricted use of the binder has been defined in a previous paper. This work shows how to extend such a calculus to multi-modal logic equipped with two features largely used in description logics, i.e. transitivity and relation inclusion assertions. An implementation of the proof procedure is also briefly presented, along with the results of some preliminary experiments.

1 Introduction

This work considers multi-modal hybrid languages (see, for instance, [3]) that, beyond the standard modalities, nominals, the satisfaction operator and the binder, include the converse modalities ($\Diamond^-_R$ and $\Box^-_R$), the global ones (E and A) and a feature largely used in description logics, i.e. the possibility of declaring an accessibility relation to be transitive and/or included in another one. Basic hybrid logic (with nominals only, beyond the modal operators $\Diamond$ and $\Box$) will be denoted by HL, and basic multi-modal hybrid logic by HL_m. Logics extending HL or HL_m with operators $O_1, \ldots, O_n$ (and their duals) are denoted by $\mathsf{HL}(O_1, \ldots, O_n)$ and $\mathsf{HL}_m(O_1, \ldots, O_n)$, respectively. Multi-modal languages including transitivity assertions and/or relation hierarchies are denoted in the same way, just including Trans (for transitivity) and/or $\sqsubseteq$ (for relation inclusion) among $O_1, \ldots, O_n$.

The satisfiability problem for formulae of any hybrid logic $\mathsf{HL}(O_1, \ldots, O_n)$ or $\mathsf{HL}_m(O_1, \ldots, O_n)$ – where $O_i \in \{@, \Diamond^-, \mathsf{E}\}$ is decidable [3]. Unfortunately, due to the high expressive power of the binder, $\mathsf{HL}(\downarrow)$ is undecidable [1, 4].

There are both semantic and syntactic restrictions allowing for regaining decidability of hybrid logic with the binder. Restricting the frame class is a way of restoring decidability, but the interplay with multi-modalities (or the addition of other operators) is not always harmless. For instance, $\mathsf{HL}(\downarrow)$ over transitive frames is decidable [18], but $\mathsf{HL}(@, \downarrow)$ and $\mathsf{HL}_m(\downarrow)$ are not [18, 17].

In [20] it is proved that the satisfiability problem for formulae in $\mathsf{HL}(@, \downarrow, \mathsf{E}, \Diamond^-)$ is decidable, provided that their negation normal form contains no universal operator (i.e. either $\Box$ or $\Box^-$ or A) scoping over a binder, that in turn has scope over a universal operator. Such a fragment of hybrid logic is denoted by $\mathsf{HL}(@, \downarrow, \mathsf{E}, \Diamond^-) \setminus \Box\downarrow\Box$. The result is proved by showing that there exists a satisfiability preserving translation of $\mathsf{HL}(@, \downarrow, \mathsf{E}, \Diamond^-) \setminus \Box\downarrow\Box$ into $\mathsf{HL}(@, \downarrow, \mathsf{E}, \Diamond^-) \setminus \downarrow\Box$, i.e.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 76–90, 2013.

the set of formulae in negation normal form where no universal operator occurs in the scope of a binder. The standard translation of hybrid logic into first order classical logic [1, 20] maps, in turn, formulae in $\mathsf{HL}(@, \downarrow, \mathsf{E}, \Diamond^-) \setminus \downarrow\Box$ into universally guarded formulae, that have a decidable satisfiability problem [12].

Decidability of $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-) \setminus \Box\downarrow\Box$ can be proved by the same reasoning, and the separate addition of either relation hierarchies or transitive relations can easily be shown to stay decidable, by reduction to the first order guarded fragment and by resorting to results already proved in the literature [19]. However, such results do not directly allow for concluding whether the logic including both features is still decidable.

This work is a continuation of previous works, where terminating tableau calculi for decidable fragments of Hybrid Logic with the binder have been defined [8, 9]. In particular, [9] presents a tableau calculus constituting a satisfiability decision procedure for $\mathsf{HL}(@, \downarrow, \mathsf{E}, \Diamond^-) \setminus \Box\downarrow\Box$. Such a procedure is here extended to multi-modal hybrid logic $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-, \mathsf{Trans}, \sqsubseteq) \setminus \Box\downarrow\Box$: a tableau calculus is presented, which terminates and is sound and complete for formulae in the fragment $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-, \mathsf{Trans}, \sqsubseteq) \setminus \downarrow\Box$, i.e. formulae in negation normal form where no occurrence of a universal operator is in the scope of a binder, with the addition of transitivity assertions and relation hierarchies. A preprocessing step along the lines of [20] turns the calculus into a satisfiability decision procedure for the fragment $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-, \mathsf{Trans}, \sqsubseteq) \setminus \Box\downarrow\Box$. Soundness, completeness and termination of the tableaux calculus thus imply that the satisfiability problem for the fragment of multi-modal hybrid logic $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-, \mathsf{Trans}, \sqsubseteq) \setminus \Box\downarrow\Box$ is decidable. The proof procedure has been implemented in a prover called Sibyl, which will be briefly presented along with the results of some preliminary experiments.

The language of $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-, \mathsf{Trans}, \sqsubseteq) \setminus \Box\downarrow\Box$ subsumes the description logic $\mathcal{SHOI}$ enriched with restricted occurrences of the binder, and allows for representing some interesting frame properties, such as, for instance, symmetry ($R^- \sqsubseteq R$), reflexivity ($\mathsf{A}\downarrow x.\Diamond_R x$), "at most" restrictions on the number of states ($\mathsf{E}\downarrow x_1. \ldots \mathsf{E}\downarrow x_n.\mathsf{A}(x_1 \vee \cdots \vee x_n)$), and "at least" restrictions on the number of R-successors of each state ($\mathsf{A}\downarrow x.\Diamond_R\downarrow y_1.(x : \Diamond_R(\neg y_1 \wedge \downarrow y_2.(x : \Diamond_R(\neg y_1 \wedge \neg y_2 \wedge \downarrow y_3. \ldots)))))$.

This section concludes with a brief introduction to the syntax and semantics of multi-modal hybrid logic with transitive relations and inclusion assertion. Well-formed expressions of $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-, \mathsf{Trans}, \sqsubseteq)$ are partitioned into two categories: *formulae* (for which the metasymbols F, G are used) and *assertions*.

Formulae are built out of a set PROP of propositional letters, a set NOM of nominals, an infinite set VAR of state variables, and a set REL of relation symbols (all such sets being mutually disjoint), and defined by the following grammar:

$$\begin{aligned} F := {} & p \mid a \mid x \mid \neg F \mid F \wedge F \mid F \vee F \mid \Diamond_R F \mid \Box_R F \\ & \mid \Diamond_R^- F \mid \Box_R^- F \mid \mathsf{E}F \mid \mathsf{A}F \mid a{:}\, F \mid x{:}\, F \mid \downarrow x.F \end{aligned}$$

where $p \in \mathsf{PROP}$, $a \in \mathsf{NOM}$, $x \in \mathsf{VAR}$ and $R \in \mathsf{REL}$. In this work, the notation $t{:}\, F$ is used (for $t \in \mathsf{NOM} \cup \mathsf{VAR}$) rather than $@_t F$. We use metavariables a, b, c for nominals, x, y, z for state variables and R, S, P for relation symbols.

If F is a formula, x a state variable and a a nominal, then $F[a/x]$ denotes the formula obtained from F by substituting a for every free occurrence of x (an occurrence of x is free if it is not in the scope of a $\downarrow x$). If $a_0, \ldots, a_n, b_0, \ldots, b_n$ are nominals, then $F[b_0/a_0, \ldots, b_n/a_n]$ denotes the formula obtained from F by simultaneously replacing b_i for every occurrence of a_i.

Assertions are either *transitivity assertions,* of the form $\mathsf{Trans}(R)$, for $R \in \mathsf{REL}$, or *inclusion assertions,* of either form $R \sqsubseteq S$ or $R^- \sqsubseteq S$, for $R, S \in \mathsf{REL}$. Here, R^- is intended to denote the inverse of the relation denoted by R, i.e. the set of pairs of states $\langle w, w' \rangle$ such that $\langle w', w \rangle$ is in the relation denoted by R. Note that inverse relations are allowed only on the left of the $\sqsubseteq$ symbol. This is only a syntactical restriction, since $R^- \sqsubseteq S^-$ is equivalent to $R \sqsubseteq S$, and $R \sqsubseteq S^-$ is equivalent to $R^- \sqsubseteq S$.

An *interpretation* $\mathcal{M}$ of an $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-, \mathsf{Trans}, \sqsubseteq)$ language is a tuple $\langle W, \rho, N, I \rangle$ where W is a non-empty set (whose elements are the *states* of the interpretation), ρ is a function mapping every $R \in \mathsf{REL}$ to a binary relation on W ($\rho(R) \subseteq W \times W$), N is a function $\mathsf{NOM} \to W$ and I a function $W \to 2^{\mathsf{PROP}}$.

If $\mathcal{M} = \langle W, \rho, N, I \rangle$ is an interpretation, $w \in W$, σ is a variable assignment for $\mathcal{M}$ (i.e. a function $\mathsf{VAR} \to W$) and F is a formula, the relation $\mathcal{M}_w, \sigma \models F$ is defined adding the following clauses to the usual definition of the classical operators:

1. $\mathcal{M}_w, \sigma \models p$ if $p \in I(w)$, for $p \in \mathsf{PROP}$.
2. $\mathcal{M}_w, \sigma \models a$ if $N(a) = w$, for $a \in \mathsf{NOM}$.
3. $\mathcal{M}_w, \sigma \models x$ if $\sigma(x) = w$, for $x \in \mathsf{VAR}$.
4. $\mathcal{M}_w, \sigma \models a{:}\, F$ if $\mathcal{M}_{N(a)}, \sigma \models F$, for $a \in \mathsf{NOM}$.
5. $\mathcal{M}_w, \sigma \models x{:}\, F$ if $\mathcal{M}_{\sigma(x)}, \sigma \models F$, for $x \in \mathsf{VAR}$.
6. $\mathcal{M}_w, \sigma \models \downarrow x.F$ if $\mathcal{M}_w, \sigma^w_x \models F$, where σ^w_x is the variable assignment such that $\sigma^w_x(x) = w$ and, for $y \neq x$, $\sigma^w_x(y) = \sigma(y)$.
7. $\mathcal{M}_w, \sigma \models \Box_R F$ if for every w' such that $\langle w, w' \rangle \in \rho(R)$, $\mathcal{M}_{w'}, \sigma \models F$.
8. $\mathcal{M}_w, \sigma \models \Diamond_R F$ if there exists w' such that $\langle w, w' \rangle \in \rho(R)$ and $\mathcal{M}_{w'}, \sigma \models F$.
9. $\mathcal{M}_w, \sigma \models \Box_{\overline{R}} F$ if for every w' such that $\langle w', w \rangle \in \rho(R)$, $\mathcal{M}_{w'}, \sigma \models F$.
10. $\mathcal{M}_w, \sigma \models \Diamond_{\overline{R}} F$ if there exists w' such that $\langle w', w \rangle \in \rho(R)$ and $\mathcal{M}_{w'}, \sigma \models F$.
11. $\mathcal{M}_w, \sigma \models \mathsf{A}F$ if $\mathcal{M}_{w'}, \sigma \models F$ for all $w' \in W$.
12. $\mathcal{M}_w, \sigma \models \mathsf{E}F$ if $\mathcal{M}_{w'}, \sigma \models F$ for some $w' \in W$.

Two formulae F and G are logically equivalent when, for every interpretation $\mathcal{M}$, assignment σ and state w of $\mathcal{M}$: $\mathcal{M}_w, \sigma \models F$ if and only if $\mathcal{M}_w, \sigma \models G$. Every formula in $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-)$ is logically equivalent to a formula in negation normal form (NNF), where negation appears only in front of atoms. Therefore, considering only formulae in NNF does not restrict the expressive power of the language.

If $\mathcal{A}$ is a set of assertions, an interpretation $\langle W, \rho, N, I \rangle$ is a model of $\mathcal{A}$ if:

1. for all $R \in \mathsf{REL}$ such that $\mathsf{Trans}(R) \in \mathcal{A}$, $\rho(R)$ is a transitive relation;
2. for all $R, S \in \mathsf{REL}$, if $R \sqsubseteq S \in \mathcal{A}$, then $\rho(R) \subseteq \rho(S)$;
3. for all $R, S \in \mathsf{REL}$ and all $w, w' \in W$, if $R^- \sqsubseteq S \in \mathcal{A}$ and $\langle w, w' \rangle \in \rho(R)$, then $\langle w', w \rangle \in \rho(S)$.

Finally, if F is a formula and $\mathcal{A}$ a set of assertions, $\{F\} \cup \mathcal{A}$ is satisfiable if there exist a model $\mathcal{M}$ of $\mathcal{A}$ and a state w of $\mathcal{M}$ such that $\mathcal{M}_w \models F$ (i.e. $\mathcal{M}_w, \sigma \models F$ for every variable assignment σ).

2 The Tableau Calculus

This section shows how to extend the system described in [9] to the presence of transitivity and inclusion assertions. The expansion rules that will be introduced to treat assertions are similar to the analogous rules presented by [13–16]. However, their addition to a terminating calculus dealing also with syntactically restricted occurrences of the binder is a novelty.

The presentation will be as self contained as possible, therefore it overlaps with the description given in [9] in many points. However, since some of the basic notions underlying the calculus are quite involved, they are not given a completely formal account.

A tableau is a set of branches, and a *tableau branch* is a sequence of *nodes* $n_0, n_1, \ldots$, where each node is labelled either by an assertion or a ground *satisfaction statement*, i.e. a formula of the form $a{:}\, F$, where no state variable occurs free in F. The nominal a in a satisfaction statement $a{:}\, F$ is called the *outermost nominal* of the formula. Node labels are always formulae in NNF. The reason why a branch is not simply a set of formulae will be briefly explained in the sequel.

If n occurs before m in a branch, we write $n < m$. The label of the node n is denoted by label(n). The notation $(n)\, a{:}\, F$ is used to denote the node n, and simultaneously say that its label is $a{:}\, F$. If a node $(n)\, a : F$ is in a branch, then the nominal a is said to label the formula F in the branch.

In order to give a more compact presentation of the expansion rules, some notions and abbreviations will be adopted. Relation symbols will also be called *forward relations* (and have *positive sign*) and the inverse of relation symbols *backward relations* (with *negative sign*). A *relation* is either a forward or backward relation. Relations are denoted by boldface letters: $\mathbf{R}$ is a meta-symbol used to denote either R itself or its inverse R^-. The following table defines some shorthands for formulae and assertions that will be used in the sequel.

$$a \Rightarrow_{\mathbf{R}} b \equiv_{def} \begin{cases} a{:}\, \Diamond_R b \text{ if } \mathbf{R} = R \\ b{:}\, \Diamond_R a \text{ if } \mathbf{R} = R^- \end{cases} \qquad a{:}\, \Diamond_{\mathbf{R}} F \equiv_{def} \begin{cases} a{:}\, \Diamond_R F \text{ if } \mathbf{R} = R \\ a{:}\, \Diamond_{\overline{R}} F \text{ if } \mathbf{R} = R^- \end{cases}$$

$$a{:}\, \Box_{\mathbf{R}} F \equiv_{def} \begin{cases} a{:}\, \Box_R F \text{ if } \mathbf{R} = R \\ a{:}\, \Box_{\overline{R}} F \text{ if } \mathbf{R} = R^- \end{cases} \qquad \mathbf{R} \sqsubseteq \mathbf{S} \equiv_{def} \begin{cases} R \sqsubseteq S & \text{if } \mathbf{R} \text{ and } \mathbf{S} \text{ have the same sign} \\ R^- \sqsubseteq S & \text{if } \mathbf{R} \text{ and } \mathbf{S} \text{ have different signs} \end{cases}$$

Let F be a ground hybrid formula in NNF and $\mathcal{A}$ a set of assertions. A tableau for $\{F\} \cup \mathcal{A}$ is initialized with a single branch, constituted by the node $(n_0)\, a_0 \colon F$, where a_0 is a new nominal, followed by nodes labelled by the assertions in $\mathcal{A}$ and then expanded according to the following *Assertion rules*:

$$\frac{}{R \sqsubseteq R}\ \mathsf{Rel}_0 \qquad \frac{\mathbf{R} \sqsubseteq \mathbf{S} \quad \mathbf{S} \sqsubseteq \mathbf{P}}{\mathbf{R} \sqsubseteq \mathbf{P}}\ \mathsf{Rel}$$

(note that Rel actually stands for four rules, according to the relation signs). Such rules complete the inclusion assertions in $\mathcal{A}$ by the reflexive and transitive closure of $\sqsubseteq$. The formula $a_0 \colon F$ is the *initial formula* of the tableau.

A tableau is expanded by application of the rules in Tables 1 and 2, which are applied to a given branch.

Table 1. Expansion rules: first group

$$\frac{(n)\, a \colon (F \wedge G)}{\begin{array}{l}(m_0)\, a \colon F \\ (m_1)\, a \colon G\end{array}}\ (\wedge) \qquad \frac{(n)\, a \colon (F \vee G)}{(m_0)\, a \colon F \quad \mid \quad (m_1)\, a \colon G}\ (\vee)$$

$$\frac{(n)\, a \colon b \colon F}{(m)\, b \colon F}\ (@) \qquad \frac{(n)\, a \colon \downarrow x.F}{(m)\, a \colon F[a/x]}\ (\downarrow)$$

$$\frac{(n)\, a \colon \Box_{\mathbf{R}} F \quad (m)\, a \Rightarrow_{\mathbf{R}} b}{(k)\, b \colon F}\ (\Box)$$

$$\frac{(n)\, a \colon \Diamond_R F}{\begin{array}{l}(m_0)\, a \colon \Diamond_R b \\ (m_1)\, b \colon F\end{array}}\ (\Diamond) \qquad \frac{(n)\, a \colon \Diamond_R^- F}{\begin{array}{l}(m_0)\, b \colon \Diamond_R a \\ (m_1)\, b \colon F\end{array}}\ (\Diamond^-)$$

where b is a fresh nominal (not applicable if F is a nominal) — where b is a fresh nominal

$$\frac{(n)\, a \colon \mathsf{A}F}{(m)\, b \colon F}\ (\mathsf{A}) \qquad \frac{(n)\, a \colon \mathsf{E}F}{(m)\, b \colon F}\ (\mathsf{E})$$

where b occurs in the branch — where b is a fresh nominal

$$\frac{\begin{array}{c}[\mathcal{B}] \\ (n)\, a \colon b\end{array}}{\mathcal{B}[b/a]}\ (=)$$

Most rules are standard, and their reading is standard too. Note that when the formulation of a rule contains (boldface) relations, it actually stands for different rules, according to the relations signs. The rules of Table 1 are the same as those presented in [9], but for the fact that the modal rules ($\Box$, $\Diamond$ and $\Diamond^-$) are here

reformulated to address the multi-modal case. The *equality rule* (=) does not add any node to the branch, but modifies the labels of its nodes. The schematic formulation of this rule in Table 1 indicates that it can be fired whenever a branch $\mathcal{B}$ contains a *nominal equality* of the form $a{:}\, b$ (with $a \neq b$); as a result of the application of the rule, every node label F in $\mathcal{B}$ is replaced by $F[b/a]$.

Formulae of the form $\Box_{\mathbf{R}} F$ and $\mathsf{A}F$ are called *universal formulae*; nodes whose labels have the form $a{:}\, G$, where G is a universal formula, are *universal nodes* and the rules $\Box$ and A are called *universal rules*. When the A rule is applied producing a node labelled by a formula of the form $b : F$, it is said to *focus* on b (and b is the focused nominal of the inference). The $\Diamond$, $\Diamond^-$ and E rules are called *blockable rules*, formulae of the form $a{:}\, \Diamond_R F$, where F is not a nominal, $a{:}\, \Diamond_{\overline{R}} F$, and $a{:}\, \mathsf{E}F$ are *blockable formulae* and a node labelled by a blockable formula is a *blockable node*. A formula of the form $a{:}\, \Diamond_R b$, where R is a forward relation, is called a *relational formula*.

The Trans rule of Table 2 deals with transitive relations and can be seen as a reformulation (in the presence of inclusion assertions) of the $\Box$ rule for transitive modal logics (a particular case of this rule is when $R = S$). In the Link rule, that deals with inclusion assertions, R is always a forward relation.

Table 2. Expansion rules: second group

$$\frac{(n)\, a{:}\, \Diamond_R b \quad (i)\, R \sqsubseteq \mathbf{S}}{(m)\, a \Rightarrow_{\mathbf{S}} b}\ (\mathsf{Link})$$

$$\frac{(n)\, a{:}\, \Box_{\mathbf{S}} F \quad (m)\, a \Rightarrow_{\mathbf{R}} b \quad (t)\, \mathsf{Trans}(R) \quad (i)\, \mathbf{R} \sqsubseteq \mathbf{S}}{(k)\, b{:}\, \Box_{\mathbf{R}} F}\ (\mathsf{Trans})$$

The premiss n of either the $\Box$ or Trans rules is called the *major premiss*, and m the *minor premiss* of the rule. In an application of the Link rule, n is the *logical premiss*. The premisses i and t, in the rules of Table 2, are the *side premisses* of the rules.

The formulation of the Trans rule is very close to the corresponding one used in description logics, where in fact "roles" include both *role names* (corresponding to relation symbols) and the inverse of role names, and inverse roles may also occur in role inclusion axioms. The abbreviation $a \Rightarrow_{\mathbf{R}} b$, however, does not have exactly the same meaning as the corresponding premiss used in the rule treating transitivity in description logics [13, 14] (a similar approach is adopted in [15]), consisting of the meta-notion "b is an $\mathbf{R}$-neighbour of a". There are two main differences between the two approaches. First of all, the semantical notion of accessibility between two states is here given a "canonical representation" in the object language (a choice already made in [8, 9]): the fact that a state a is R-related to b is represented by the *relational formula* $a{:}\, \Diamond_R b$. Though semantically equivalent to $b{:}\, \Diamond_{\overline{R}} a$, the latter is not a relational formula, i.e. it is not the

canonical representation of an R-relation. This is reflected by the fact that the $\Diamond$ rule cannot be applied to a relational formula, while $b\colon \Diamond_R^- a$ can be expanded by means of the $\Diamond^-$ rule. Moreover, in the present work, the notation $a \Rightarrow_{\mathbf{R}} b$ is only an abbreviation for a relational formula, which does not take subrelations into account: it may be the case that $a \Rightarrow_{\mathbf{S}} b$ belongs to a given branch $\mathcal{B}$ for some $\mathbf{S} \sqsubseteq \mathbf{R}$, and yet $a \Rightarrow_{\mathbf{R}} b$ does not. The fact that, in the present work, no meta-notion is used to represent "$\mathbf{R}$-neighbours" is responsible for the presence of the Link rules, that have no counterpart in [13–15].

The first node of a branch $\mathcal{B}$ is called the *top node* and its label the *top formula* of $\mathcal{B}$. Nominals occurring in the top formula are called *top nominals*. The notion of top nominal is relative to a tableau branch, because applications of the equality rule may change the top formula, hence the set of top nominals.

A branch is *closed* whenever it contains, for some nominal a, either a pair of nodes $(n)\, a\colon p$, $(m)\, a\colon \neg p$ for some $p \in \mathsf{PROP}$, or a node $(n)\, a\colon \neg a$. As usual, it is assumed that a closed branch is never expanded further. A branch which is not closed is *open*. A branch is *complete* when it cannot be further expanded.

Provided that the initial formula is in $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-) \setminus \downarrow\Box$, the calculus enjoys the following important *strong subformula property*, used to prove both termination and completeness: every universal formula occurring in a tableau branch is obtained from a subformula of the top formula F_0 of the branch by possibly replacing operators $\Box_{\mathbf{R}}$ with $\Box_{\mathbf{S}}$, for some relation $\mathbf{S}$ in the language of F_0. Treating nominal equalities by means of substitution, like in [6, 7, 9, 11], is essential to ensure such a property. By the effect of substitution, however, distinct node labels may become equal, though the corresponding nodes are still distinct elements of the branch.

The reason why nodes with the same label do not collapse is that they must be arrangeable in a tree-like structure, where each node has at most one parent. The relation on nodes inducing such a structure (see Definition 2) is used to define indirect blocking (Definition 3). Termination is in fact achieved by means of a form of anywhere blocking with indirect blocking.

Direct blocking is a relation between nodes in a tableau branch, holding whenever the respective labels (formulae) are equal up to (a proper form of) nominal renaming. Essentially, in order for a node $(n)\, F$ to (directly) block $(m)\, G$ in a branch $\mathcal{B}$, it must be the case that $G = F[a_1/b_1, \ldots, a_n/b_n]$, where $a_1, \ldots, a_n$, $b_1 \ldots, b_n$ are *non-top* nominals such that, for all $i = 1, \ldots, n$, a_i and b_i label the same set of propositions in PROP and the same formulae of the form $\Box_{\mathbf{R}} F$. More precisely:

Definition 1 (Nominal compatibility and mappings). *If $\mathcal{B}$ is a tableau branch, then:*

1. *two nominals a and b are* compatible *in $\mathcal{B}$ if they label the same propositions in* PROP *and the same formulae of the form $\Box_{\mathbf{R}} F$.*
2. *A mapping π for $\mathcal{B}$ is an* injective *function from* non-top *nominals to* non-top *nominals such that for all a, a and $\pi(a)$ are compatible in $\mathcal{B}$. Mappings*

are extended to act on formulae in the obvious way: $\pi(F)$ is the formula obtained by substituting $\pi(a)$ for a in F, for every nominal a.

3. *A mapping π for $\mathcal{B}$ maps a formula F to a formula G if $\pi(F) = G$ and π is the identity for all nominals which do not occur in F.*
4. *A formula F can be mapped to a formula G in $\mathcal{B}$ if there exists a mapping π for $\mathcal{B}$ mapping F to G.*

The (direct) blocking restriction forbids the application of a blockable rule to a node n, whenever the label of a node $m < n$ can be mapped to label(n).

As already mentioned before, indirect blocking relies on a partial order on the nodes of a branch $\mathcal{B}$, called the *offspring relation* and denoted by $\prec_{\mathcal{B}}$, which arranges them into a family of trees, where non-terminal nodes are blockable nodes. Every tree is rooted at a node called a *root node* (a node with no *parents* w.r.t. the offspring relation). When a blockable rule is applied, the generated nodes are *children* of the expanded node. All the other rules generate *siblings* of one of the premisses of the inference (two nodes are siblings either if they are both root nodes or they have the same parent).

Properly, the offspring relation and blockings are defined by a mutual recursion on branch construction: if $\mathcal{B}'$ is a branch obtained by expanding $\mathcal{B}$, the definition of $\prec_{\mathcal{B}'}$ assumes that the set of blocked nodes in $\mathcal{B}$ is already defined, and indirectly blocked nodes in $\mathcal{B}$ depend on the relation $\prec_{\mathcal{B}}$. This is due to the presence of the A rule, for which a *minor premiss* must be defined, since nodes added to a branch $\mathcal{B}$ by an application $\mathcal{I}$ of the A rule are siblings of such a minor premiss (in the new branch $\mathcal{B}'$ obtained from the expansion); but, in order to determine the minor premiss of $\mathcal{I}$ it is necessary to know which nodes are blocked in $\mathcal{B}$.

The presentation that follows is somewhat simplified, and the reader is referred to [9] for the more formal approach. Let us assume that when the A rule is applied, beyond the premiss shown in Table 1, the branch contains a node called the minor premiss of the rule application (which will be defined further on, in Definition 5).

Definition 2 (Offspring relation). *Let $\mathcal{B}$ be a tableau branch.*

1. *Every node already contained in the initial branch from which $\mathcal{B}$ is obtained (i.e. its top node and all the nodes labelled by assertions) is a root node.*
2. *If a node n has been added to $\mathcal{B}$ by application of a blockable rule to node m, then $m \prec_{\mathcal{B}} n$ (n is a child of m and m is the parent of n).*
3. *If n has been added to $\mathcal{B}$ by application of either a universal rule or the* Trans *rule, whose minor premiss is m, then n is a sibling of m (i.e., if m is a root node, then n is a root node too; otherwise, if $k \prec_{\mathcal{B}} m$, then $k \prec_{\mathcal{B}} n$).*
4. *If n has been added to $\mathcal{B}$ by application of any other rule of table 1 (i.e. any other single-premiss rule) to node m, then n is a sibling of m.*
5. *If n has been added to $\mathcal{B}$ by application of the* Link *rule, then n is a sibling of the logical premiss of the inference.*

It is worth pointing out that an application of either the Trans rule or a universal one produces a sibling of the minor premiss of the inference, and not the

major one. This is an essential feature of the offspring relation, needed to prove termination.

The notions of direct and indirect blocking can now be defined.

Definition 3 (Direct and indirect blocking). *Let $\mathcal{B}$ be a tableau branch. The set of directly and indirectly blocked nodes in $\mathcal{B}$ is defined by induction on the (total) order $<$ on the nodes of $\mathcal{B}$:*

- *n is blocked if it is either directly or indirectly blocked.*
- *n is directly blocked by m if n is a blockable node, $m < n$, m is not blocked and* label(m) *can be mapped to* label(n) *in $\mathcal{B}$; n is directly blocked in $\mathcal{B}$ if it is directly blocked by some m in $\mathcal{B}$.*
- *n is indirectly blocked if it is not directly blocked and it has an ancestor w.r.t. $\prec_{\mathcal{B}}$ which is blocked.*

An indirectly blocked node is called a phantom node *(or, simply, a phantom).*

It is worth noticing that a node is a phantom if and only if all its siblings are phantoms too.

The application of the expansion rules is restricted by the conditions defined next. Restrictions **R1**–**R4** are essentially the same as those formulated in [9]. The restrictions concerning the new rules are formulated apart (**R5**–**R6**).

Definition 4 (Restrictions on the expansion rules). *The expansion of a tableau branch $\mathcal{B}$ is subject to the following restrictions:*

R1. *no node labelled by a formula already occurring in $\mathcal{B}$ as the label of a* non-phantom *node is ever added to $\mathcal{B}$.*
R2. *Blockable nodes can be expanded at most once in a branch.*
R3. *A phantom node cannot be expanded by means of a single-premiss rule (including the equality rule), nor can it be used as the minor premiss of a universal rule.*
R4. *A blockable node n cannot be expanded if it is directly blocked in $\mathcal{B}$.*
R5. *A phantom node cannot be used as the minor premiss of the* Trans *rule.*
R6. *A phantom node cannot be used as the logical premiss of the* Link *rule.*

Finally, we only need to define the minor premiss of an application of the A rule.

Definition 5. *If $\mathcal{B}$ is obtained from $\mathcal{B}'$ by means of an application $\mathcal{I}$ of the A rule focusing on the nominal b, then the minor premiss of $\mathcal{I}$ is the first non-phantom node in $\mathcal{B}'$ where b occurs.*

Note that, as a particular case of restriction **R3**, the A rule cannot focus on a nominal which only occurs in phantom nodes in the branch. Consequently, thanks to restriction **R3**, every application of the A rule has a minor premiss.

Due to space restrictions, the termination and completeness proofs cannot be included in this work, but can be found in [10]. Here, only a short proof sketch is included.

Theorem 1 (Termination). *If the initial formula of a tableau is in the fragment* $\mathsf{HL}(@, \downarrow, \mathsf{E}, \Diamond^-) \setminus \downarrow\Box$*, then every tableau branch has a bounded depth and tableau construction always terminates.*

Termination if proved by showing that the nodes of a branch $\mathcal{B}$ are arranged by the offspring relation into a bounded sized set of trees, each of which has bounded width and bounded depth. This holds because a branch is not a set of formulae, but nodes, and each node has at most one parent. If nodes labelled by the same formula collapsed into a single branch element, such an element might have multiple parents.[1]

The drawback is that the reasoning proving that any node has a bounded number of siblings is not as simple as it would be if dealing with sets of formulae. It relies in an essential way on the fact that universal rules do not generate siblings of their major premisses and, thanks to the mentioned strong subformula property, the number of universal formulae occurring in a tableau branch is bounded.

In order to prove that tree depth is also bounded, it is shown that the size of any set of blockable nodes which may occur in a tableau branch, and such that none of its elements blocks another one, is bounded. This holds for two reasons. First of all, the calculus enjoys a *weak subformula property*: for any non-relational formula $a\colon F$ occurring in a tableau branch, F is obtained from a subformula of the top formula F_0 of the branch by replacing free variables with nominals and, possibly, operators $\Box_{\mathbf{R}}$ with $\Box_{\mathbf{S}}$, for some relation $\mathbf{S}$ in the language of F_0. Secondly, the strong subformula property ensures that the number of nominal compatibility classes is bounded.

Theorem 2 (Completeness). *Let F be a formula and $\mathcal{A}$ a set of assertions. If $\{F\} \cup \mathcal{A}$ is in $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-, \mathsf{Trans}, \sqsubseteq) \setminus \downarrow\Box$ and is unsatisfiable, then any complete tableau for $\{F\} \cup \mathcal{A}$ is closed.*

In order to prove that the calculus is complete, it is shown – like in [9] – how to extend a subset $\mathcal{N}_0$ of any complete and open branch $\mathcal{B}$ in such a way that every directly blocked node is added a suitable "witness" (the witness(es) of a blockable node n can be viewed simply as node(s) which could by obtained by application of the corresponding blockable rule to n). The fact that the labels of blocked and blocking nodes are not necessarily identical does not allow taking the witness of the blocking node as a witness of the blocked one. Nor can a model be simply built from a set of states consisting of equivalence classes of nominals, where two nominals are in the same class whenever some blocking mapping maps one to the other: two nominals a and b may be compatible even if the branch contains a node labelled by $a\colon \neg b$.

The initial set of the construction, $\mathcal{N}_0$, is the union of the non-phantom nodes in $\mathcal{B}$ and the nodes of the form $(n)\, a\colon F$, with a occurring in some non-phantom node in $\mathcal{B}$ and either F has the form $\Box_{\mathbf{R}} G$ or $F \in \mathsf{PROP}$. $\mathcal{N}_0$ is extended by steps,

[1] For a similar reason it is not possible to block nominals instead of nodes: two nominals with different parents may become equal by substitution.

constructing a (possibly infinite) sequence of sets of nodes $\mathcal{N}_0 \subseteq \mathcal{N}_1 \subseteq \mathcal{N}_2 \ldots$, where each $\mathcal{N}_{i+1}$ is obtained from $\mathcal{N}_i$ by (fairly) choosing a blockable node n with no witness in $\mathcal{N}_i$. The construction ensures that there exists a node $n_0 \in \mathcal{N}_0$ whose label can be mapped to label(n) in $\mathcal{N}_i$. The blocking mapping is then used to add new nodes and obtain $\mathcal{N}_{i+1}$, in such a way that n has a witness in $\mathcal{N}_{i+1}$. It is finally shown how to build a model of the initial formula from the union of the sets $\mathcal{N}_i$ (due to the presence of assertions, the construction is quite different from the corresponding one in [9]).

We conclude with some examples illustrating the calculus in action. The first simple one below shows the interplay between the Trans and Link rules. It consists of the closed one-branch tableau represented below for the formula $\Diamond_S \Diamond_S p \wedge \Box_S \neg p$, together with the assertions $\mathsf{Trans}(R)$, $R \sqsubseteq S$, $S \sqsubseteq R$. The notations $n \rightsquigarrow^{\mathcal{R}} m$ or $(n_1, \ldots, n_k) \rightsquigarrow^{\mathcal{R}} m$, used in the rightmost column below, means that the addition of node m is due to the application of rule $\mathcal{R}$ to node n (or nodes $n_1, \ldots, n_k$). Nodes 0–4 constitute the initial tableau. The branch is closed because of nodes 11 and 15.

(0) $a\colon (\Diamond_S \Diamond_S p \wedge \Box_S \neg p)$		(8) $a\colon \Diamond_S b$	$6 \rightsquigarrow^{\Diamond} 8$
(1) $\mathsf{Trans}(R)$		(9) $b\colon \Diamond_S p$	$6 \rightsquigarrow^{\Diamond} 9$
(2) $R \sqsubseteq S$		(10) $b\colon \Diamond_S c$	$9 \rightsquigarrow^{\Diamond} 10$
(3) $S \sqsubseteq R$		(11) $c\colon p$	$9 \rightsquigarrow^{\Diamond} 11$
(4) $R \sqsubseteq R$	Rel_0	(12) $a\colon \Diamond_R b$	$(8,3) \rightsquigarrow^{\mathsf{Link}} 12$
(5) $S \sqsubseteq S$	Rel_0	(13) $b\colon \Diamond_R c$	$(10,3) \rightsquigarrow^{\mathsf{Link}} 13$
(6) $a\colon \Diamond_S \Diamond_S p$	$0 \rightsquigarrow^{\wedge} 6$	(14) $b\colon \Box_R \neg p$	$(7,12,1,2) \rightsquigarrow^{\mathsf{Trans}} 14$
(7) $a\colon \Box_S \neg p$	$0 \rightsquigarrow^{\wedge} 7$	(15) $c\colon \neg p$	$(14,13) \rightsquigarrow^{\Box} 15$

Next example illustrates the dynamic nature of blockings. Figure 1 represents a complete and open tableau branch $\mathcal{B}$ for the formula $F = (\mathsf{A}{\downarrow}x.\Diamond_{R^-} \Diamond_R \neg x) \wedge \Box_R p$ – which holds in a state w if every state of the interpretation has at least one R-sibling, and p holds in every state R-related to w – where R is a transitive relation. In the representation of the branch given below, $G = \Diamond_{R^-} \Diamond_R \neg x$ and, in the notation $(n, m) \rightsquigarrow^{\mathsf{A}} k$, n is the major premiss of the inference and m the minor one.

The relation $\prec_{\mathcal{B}}$ in this branch can be described as follows, where the notation $n \prec_{\mathcal{B}} \{m_1, \ldots, m_k\}$ abbreviates $n \prec_{\mathcal{B}} m_1$ and $\ldots n \prec_{\mathcal{B}} m_k$. Nodes $0 \ldots 6$ are root nodes, and $6 \prec_{\mathcal{B}} \{7, 8, 9, 12, 16, 18\}$, $8 \prec_{\mathcal{B}} \{10, 11, 13, 14, 15, 17\}$, $17 \prec_{\mathcal{B}} \{19, 20, 23, 33, 35\}$, $18 \prec_{\mathcal{B}} \{21, 22, 26, 31\}$, $20 \prec_{\mathcal{B}} \{24, 25, 29, 30, 32, 34\}$, $22 \prec_{\mathcal{B}} \{27, 28\}$, $35 \prec_{\mathcal{B}} \{36, 37, 38, 41\}$, $37 \prec_{\mathcal{B}} \{39, 40\}$. For instance, node 7 is the minor premiss of the application of the $\Box$ rule producing 12, and 10 is the minor premiss of the application of the Trans rule producing 13, therefore 7 and 12 are siblings and so are 10 and 13. When the A rule is applied to produce node 15 focusing on the nominal a_3, the first non-phantom node where a_3 occurs is 10, so that 10 is the minor premiss of the inference and a sibling of 15.

In order to illustrate blockings, the notation $\mathcal{B}_n$ is used to denote the branch segment up to node n, and $a_i \approx_n a_j$ means that a_i and a_j are compatible in $\mathcal{B}_n$ (note that, in this example, the formulae to be taken into account to check compatibilities are p and $\Box_R p$). Node 17 cannot be blocked by 6, and 20 cannot

0) $a_1 : F$
1) $\mathsf{Trans}(R)$
2) $R \sqsubseteq R$
3) $a_1 : \mathsf{A}{\downarrow}x.G$ $0 \rightsquigarrow^{\wedge} 3$
4) $a_1 : \Box_R p$ $0 \rightsquigarrow^{\wedge} 4$
5) $a_1 : {\downarrow}x.G$ $(3, 0) \rightsquigarrow^{\mathsf{A}} 5$
6) $a_1 : \Diamond_{R^-} \Diamond_R \neg\, a_1$ $5 \rightsquigarrow^{\downarrow} 6$
7) $a_2 : \Diamond_R a_1$ $6 \rightsquigarrow^{\Diamond} 7$
8) $a_2 : \Diamond_R \neg\, a_1$ $6 \rightsquigarrow^{\Diamond} 8$
9) $a_2 : \Box_R p$ $(4, 7, 1, 2) \rightsquigarrow^{\mathsf{Trans}} 9$
10) $a_2 : \Diamond_R a_3$ $8 \rightsquigarrow^{\Diamond} 10$
11) $a_3 : \neg\, a_1$ $8 \rightsquigarrow^{\Diamond} 11$
12) $a_1 : p$ $(9, 7) \rightsquigarrow^{\Box} 12$
13) $a_3 : \Box_R p$ $(9, 10, 1, 2) \rightsquigarrow^{\mathsf{Trans}} 13$
14) $a_3 : p$ $(9, 10) \rightsquigarrow^{\Box} 14$
15) $a_3 : {\downarrow}x.G$ $(3, 10) \rightsquigarrow^{\mathsf{A}} 15$
16) $a_2 : {\downarrow}x.G$ $(3, 7) \rightsquigarrow^{\mathsf{A}} 16$
17) $a_3 : \Diamond_{R^-} \Diamond_R \neg\, a_3$ $15 \rightsquigarrow^{\downarrow} 17$
18) $a_2 : \Diamond_{R^-} \Diamond_R \neg\, a_2$ $16 \rightsquigarrow^{\downarrow} 18$
19) $a_4 : \Diamond_R a_3$ $17 \rightsquigarrow^{\Diamond} 19$
20) $a_4 : \Diamond_R \neg\, a_3$ $17 \rightsquigarrow^{\Diamond} 20$
21) $a_5 : \Diamond_R a_2$ $18 \rightsquigarrow^{\Diamond} 21$
22) $a_5 : \Diamond_R \neg\, a_2$ $18 \rightsquigarrow^{\Diamond} 22$
23) $a_4 : \Box_R p$ $(13, 19, 1, 2) \rightsquigarrow^{\mathsf{Trans}} 23$
24) $a_4 : \Diamond_R a_6$ $20 \rightsquigarrow^{\Diamond} 24$
25) $a_6 : \neg\, a_3$ $20 \rightsquigarrow^{\Diamond} 25$
26) $a_5 : \Box_R p$ $(9, 21, 1, 2) \rightsquigarrow^{\mathsf{Trans}} 26$
27) $a_5 : \Diamond_R a_7$ $22 \rightsquigarrow^{\Diamond} 27$
28) $a_7 : \neg\, a_2$ $22 \rightsquigarrow^{\Diamond} 28$
29) $a_6 : \Box_R p$ $(23, 24, 1, 2) \rightsquigarrow^{\mathsf{Trans}} 29$
30) $a_6 : p$ $(23, 24) \rightsquigarrow^{\Box} 30$
31) $a_2 : p$ $(26, 21) \rightsquigarrow^{\Box} 31$
32) $a_6 : {\downarrow}x.G$ $(3, 24) \rightsquigarrow^{\mathsf{A}} 32$
33) $a_4 : {\downarrow}x.G$ $(3, 19) \rightsquigarrow^{\mathsf{A}} 33$
34) $a_6 : \Diamond_{R^-} \Diamond_R \neg\, a_6$ $32 \rightsquigarrow^{\downarrow} 34$
35) $a_4 : \Diamond_{R^-} \Diamond_R \neg\, a_4$ $33 \rightsquigarrow^{\downarrow} 35$
36) $a_8 : \Diamond_R a_4$ $35 \rightsquigarrow^{\Diamond} 36$
37) $a_8 : \Diamond_R \neg\, a_4$ $35 \rightsquigarrow^{\Diamond} 37$
38) $a_8 : \Box_R p$ $(23, 36, 1, 2) \rightsquigarrow^{\mathsf{Trans}} 38$
39) $a_8 : \Diamond_R a_9$ $37 \rightsquigarrow^{\Diamond} 39$
40) $a_9 : \neg\, a_4$ $37 \rightsquigarrow^{\Diamond} 40$
41) $a_4 : p$ $(38, 36) \rightsquigarrow^{\Box} 41$

Fig. 1. A complete tableau branch for $\{(\mathsf{A}{\downarrow}x.\Diamond_{R^-} \Diamond_R \neg\, x) \wedge \Box_R p, \mathsf{Trans}(R)\}$

be blocked by 8, because a_1 is a top nominal and mappings can only affect non-top ones. In the whole branch $\mathcal{B}$, the nodes 18, 34 and 35 are blocked by 17 (note that 17 is not an ancestor of 18), because $a_3 \approx_{41} a_2 \approx_{41} a_4 \approx_{41} a_6$. Their descendants $(21, 22, 26 - 28, 31, 36 - 41)$ are therefore phantoms in $\mathcal{B}$. However, while 34 is not expanded because it is blocked by 17 in $\mathcal{B}_{34}$ (because $a_3 \approx_{34} a_6$), 18 is not blocked in $\mathcal{B}_i$ for all $i < 31$, i.e. until $a_2 : p$ is added to the branch. Therefore 18 is expanded. Analogously, 35 is not blocked by 17 until $a_4 : p$ is added to the branch (node 41). The branch is complete: every non blocked node has been expanded or used as the minor premiss of a suitable rule. In particular, note that the nominals a_5, a_7, a_8, a_9 occur only in phantom nodes, therefore the A rule cannot focus on them.

3 The Sibyl Prover

The calculus described in Section 2 has been implemented in a prover called Sibyl, that is available at `http://cialdea.dia.uniroma3.it/sibyl/`. It is written in Objective Caml and takes as input a file containing a set of assertions and a set of formulae, checks them for satisfiability and outputs the result. Every input formula in $\mathsf{HL}_m(@, {\downarrow}, \mathsf{E}, \Diamond^-) \setminus \Box{\downarrow}\Box$ is preprocessed and translated into the fragment $\mathsf{HL}_m(@, {\downarrow}, \mathsf{E}, \Diamond^-) \setminus {\downarrow}\Box$, by use of the satisfiability preserving translation defined in [20]. If some formula is not in $\mathsf{HL}_m(@, {\downarrow}, \mathsf{E}, \Diamond^-) \setminus \Box{\downarrow}\Box$, then Sibyl warns the user that termination and correctness of the result are not guaranteed. At

present, backjumping is the only important optimization technique implemented in the prover.

In order to test Sibyl for correctness, it could not be compared to other provers for modal or description logics, since, to the author's knowledge, the hybrid binder and relation hierarchies coexist in none of them. For the same reason it would not make much sense using problems in existing repositories for modal or description logic. Therefore Sibyl has been run on a set of randomly generated tests, and the translations of the same tests into first order logic (using the standard translation of hybrid logic formulae and the straightforward translation of assertions) have then been given in input to the SPASS prover [21]. Each test is based on a file generated by hGen [2], modified so as to obtain formulae in $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-) \setminus \Box\downarrow\Box$ and with the addition of a random set of transitivity and inclusion assertion. A first group of 1620 tests has been generated with 30% probability for a relation to be transitive and 30% probability for any pair of relations R, S to be related by either $R \sqsubseteq S$ or $R^- \sqsubseteq S$. The tests are grouped according to their modal degree (varying from 2 to 10), each group containing tests with 10 to 50 clauses (hGen generates sets of clauses). In order to evaluate the impact of the presence of assertions on Sibyl's behaviour, other four groups of tests have been obtained from the basic set, reducing the number of assertions in each file, respectively to 75%, 50%, 25% and no assertions at all.

Sibyl and SPASS have been run on these test sets with one minute timeout and they agree on the outcome of all problems where both provers terminate successfully. The test sets, the detailed results of the experiments and diagrams summarizing them can be downloaded from Sibyl web page.

Though the experiments only aimed at testing Sibyl for correctness, they were also an opportunity to give a preliminary evaluations of its performances compared to SPASS (that was run in default mode, since, from some preliminary tests, other flag settings appeared either to degrade its performance or have no significant effect). Quite surprisingly, although SPASS is a mature prover and Sibyl a newborn, the latter turned out to globally outperform the former. SPASS could not solve about 13% of the problems in the allowed one minute time, while Sibyl failed in less than 5.5%. Taking the number of timeouts as a performance measure, the impact of the number of assertions and the modal degree of formulae has been evaluated. In the tests with no assertions SPASS performs better than Sibyl: 2.22% timeouts versus Sibyl's 4.81%. On the other hand, SPASS could not solve 21.98% tests of the base set (with no reduction of the number of assertions), while Sibyl 6.30%. With respect to the effect of the modal degree on the behaviour of the provers, in the base set, for instance, SPASS ran out of time in 2.22% tests of modal degree 2, and it reached 32.22% timeouts in the problems of modal depth 10. In the same set of problems, Sibyl's failures range from 6.67% (modal degree 2) to 9.44% (modal degree 10).

The experimental results show that Sibyl's behaviour only slightly degrades when the number of assertions and the modal degree increase. In comparison, the first order prover appears to be much more sensitive to the number of assertions, especially when the modal degree becomes higher. Presumably, this is not

a credit to Sibyl, but rather an instance of the poor behaviour exhibited by first order theorem provers when fed with non optimized translations of modal formulae. In order to refine such a preliminary analysis, other encoding principles should be used and tested, and the effect of transitivity and inclusion assertions should be analysed separately.

4 Concluding Remarks

This work presents a satisfiability decision procedure for hybrid formulae in $\mathsf{HL}_m(@, \downarrow, \mathsf{E}, \Diamond^-, \mathsf{Trans}, \sqsubseteq) \setminus \Box\downarrow\Box$, and its implementation in the Sibyl prover. Transitivity and relation inclusion assertions are treated by expansion rules which are very close to (though not exactly the same as) the analogous rules presented in [13–16]. The main result of this work is proving that they can be added to a calculus dealing also with restricted occurrences of the binder, maintaining termination, beyond soundness and completeness.

Differently from other terminating tableau calculi for (binder-free) hybrid logic including the global and converse modalities, blocking concerns here nodes (corresponding to formulae) and not nominals (i.e. sets of formulae). In the absence of the binder, compatibility checks, requiring to exit from the "local" view and look for other formulae in the branch, are needed only for the formulae outermost nominals and concern only a subset of the formulae labelled by such nominals. Indirect blocking, in turn, relies on a particular partial order on nodes, arranging them in a family of trees of bounded width and bounded depth. Width boundedness is guaranteed by the fact that universal nodes (which may be expanded a potentially unbounded number of times) do not generate "siblings".

Other works have addressed the issue of representing frame properties and/or relation hierarchies in tableau calculi for binder-free hybrid logic (for instance, [5, 15, 16]). The maybe richer calculus of this kind is [15], that considers graded and global modalities, reflexivity, transitivity and role hierarchies. The converse modalities are however missing, and inverse relations are not allowed.

The possibility of adding graded modalities (i.e. number restrictions of description logics) to the calculus presented in this work is an interesting but hard issue. As a matter of fact, whether restricted occurrences of the binder can coexist with graded modalities in a decidable hybrid logic is an open question.

Acknowledgments. The author's implementation (and debugging) work has built upon the bachelor or master projects of several students. Beyond those who worked on Herod [11], Sibyl's ancestor, the author is especially indebted to Giulia Di Rienzo, who implemented Sibyl's first version.

References

1. Areces, C., Blackburn, P., Marx, M.: A road-map on complexity for hybrid logics. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 307–321. Springer, Heidelberg (1999)

2. Areces, C., Heguiabehere, J.: hGen: A random CNF formula generator for hybrid languages. In: Methods for Modalities 3 (M4M-3), Nancy, France (2003)
3. Areces, C., ten Cate, B.: Hybrid logics. In: Handbook of Modal Logics, pp. 821–868. Elsevier (2007)
4. Blackburn, P., Seligman, J.: Hybrid languages. Journal of Logic, Language and Information 4, 251–272 (1995)
5. Bolander, T., Blackburn, P.: Terminating tableau calculi for hybrid logics extending K. Electronic Notes in Theoretical Computer Science 231, 21–39 (2009)
6. Cerrito, S., Cialdea Mayer, M.: An efficient approach to nominal equalities in hybrid logic tableaux. Journal of Applied Non-classical Logics 20(1-2), 39–61 (2010)
7. Cerrito, S., Cialdea Mayer, M.: Nominal substitution at work with the global and converse modalities. In: Advances in Modal Logic, vol. 8, pp. 57–74. College Publications (2010)
8. Cerrito, S., Cialdea Mayer, M.: A tableaux based decision procedure for a broad class of hybrid formulae with binders. In: Brünnler, K., Metcalfe, G. (eds.) TABLEAUX 2011. LNCS, vol. 6793, pp. 104–118. Springer, Heidelberg (2011)
9. Cerrito, S., Cialdea Mayer, M.: A tableau based decision procedure for a fragment of hybrid logic with binders. Journal of Automated Reasoning (2012) (published online, to appear on paper)
10. Cialdea Mayer, M.: Tableaux for multi-modal hybrid logic with binders, transitive relations and relation hierarchies. Technical Report RT-DIA-199-2012, Dipartimento di Informatica e Automazione, Università di Roma Tre (2012)
11. Cialdea Mayer, M., Cerrito, S.: Herod and Pilate: two tableau provers for basic hybrid logic. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 255–262. Springer, Heidelberg (2010)
12. Grädel, E.: On the restraining power of guards. Journal of Symbolic Logic 64, 1719–1742 (1998)
13. Horrocks, I., Sattler, U.: A description logic with transitive and inverse roles and role hierarchies. Journal of Logic and Computation 9(3), 385–410 (1999)
14. Horrocks, I., Sattler, U.: A tableau decision procedure for $\mathcal{SHOIQ}$. Journal of Automated Reasoning 39(3), 249–276 (2007)
15. Kaminski, M., Schneider, S., Smolka, G.: Terminating tableaux for graded hybrid logic with global modalities and role hierarchies. Logical Methods in Computer Science 7(1) (2011)
16. Kaminski, M., Smolka, G.: Terminating tableau systems for hybrid logic with difference and converse. Journal of Logic, Language and Information 18(4), 437–464 (2009)
17. Mundhenk, M., Schneider, T.: Undecidability of multi-modal hybrid logics. Electronic Notes in Theoretical Computer Science 174(6), 29–43 (2007)
18. Mundhenk, M., Schneider, T., Schwentick, T., Weber, V.: Complexity of hybrid logics over transitive frames. Journal of Applied Logic 8(4), 422–440 (2010)
19. Szwast, W., Tendera, L.: On the decision problem for the guarded fragment with transitivity. In: Proc. of the 16th Symposium on Logic in Computer Science (LICS), pp. 147–156 (2001)
20. ten Cate, B., Franceschet, M.: On the complexity of hybrid logics with binders. In: Ong, L. (ed.) CSL 2005. LNCS, vol. 3634, pp. 339–354. Springer, Heidelberg (2005)
21. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 140–145. Springer, Heidelberg (2009)

Tractable Inference Systems: An Extension with a Deducibility Predicate

Hubert Comon-Lundh[1], Véronique Cortier[2], and Guillaume Scerri[1,2]

[1] LSV, CNRS & ENS Cachan, France
[2] LORIA, CNRS, France

Abstract. The main contribution of the paper is a PTIME decision procedure for the satisfiability problem in a class of first-order Horn clauses. Our result is an extension of the tractable classes of Horn clauses of Basin & Ganzinger in several respects. For instance, our clauses may contain atomic formulas $S \vdash t$ where $\vdash$ is a predicate symbol and S is a finite set of terms instead of a term. $\vdash$ is used to represent any possible computation of an attacker, given a set of messages S. The class of clauses that we consider encompasses the clauses designed by Bana & Comon-Lundh for security proofs of protocols in a computational model.

Because of the (variadic) $\vdash$ predicate symbol, we cannot use ordered resolution strategies only, as in Basin & Ganzinger: given $S \vdash t$, we must avoid computing $S' \vdash t$ for all subsets S' of S. Instead, we design PTIME entailment procedures for increasingly expressive fragments, such procedures being used as oracles for the next fragment.

Finally, we obtain a PTIME procedure for arbitrary ground clauses and saturated Horn clauses (as in Basin & Ganzinger), together with a particular class of (non saturated) Horn clauses with the $\vdash$ predicate and constraints (which are necessary to cover the application).

1 Introduction

1.1 The Application Context

The design of automated security proofs is a topic extensively studied for over 20 years. One problem that was raised about 12 years ago is the validity (or the scope) of such proofs. More specifically, for most of the automatic security proofs messages are abstracted by terms and the attackers capabilities are restricted to a specific set of operations. In contrast, modern cryptography typically considers attackers that can perform any computation that does not require too much time (say, in probabilistic polynomial time). This includes of course some computations that are not explicitly specified. This issue has been first addressed by M. Abadi and P. Rogaway [1], followed by many authors. The idea is to prove that the symbolic formal model is *sound* with respect to the more concrete computational model: if there is no attack in the symbolic model, then there is no attack in the computational model, except with negligible probability. There are several such soundness proofs, for various primitives and in various contexts (see

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 91–108, 2013.

e.g. [11,2,9] to cite only a few). However, all these results require heavy proofs and assume strong hypotheses, some of which are not quite realistic. Typical examples of unrealistic assumptions include: a key cycle is never created, or the attacker does use his own keys.

These difficulties lead to try to prove the security protocols directly in the computational model. For instance CryptoVerif [7] or Easycrypt [5] are designed in this spirit. The proofs have however to account for probability distributions computations, attacker's time computation, and are relatively difficult, often requiring user interactions. We study here an alternative approach presented in [4] which consists in specifying formally what the attacker *cannot do*. Each axiom in such a specification can be a consequence of an assumption on the primitives, which yields the soundness of the model by construction. The drawback is however the proof automation in this model: there was no evidence that this is possible in a reasonably efficient way. This is the problem that we want to address in this paper.

In the model of [4], transitions of the system are possible, as soon as they do not contradict the axioms. Hence, an attack consists in a sequence of attacker's actions, that is consistent with the axioms and the negation of the security property. Conversely, if all (symbolic) transition sequences yield a formula, which is inconsistent with the axioms and the negation of the security property, then the protocol is secure, for any attacker, in any model that satisfies the axioms. The clauses make use of a *deducibility predicate* $\vdash$, whose interpretation is not fixed: it stands for any attacker's computation. In other words, $S \vdash h$ states that the attacker must be able to compute h from his knowledge at this stage.

In summary, checking for cryptographic security amounts to checking the satisfiability of a finite set of ground formulas Φ together with axioms A (which are Horn clauses) and the negation of the security property π (a ground fact). Since, in practice, this satisfiability check has to be performed for any interleaving of (symbolic) actions, it must be efficiently performed. Fortunately, the formulas are not arbitrary first-order formulas. We introduce them informally below.

- Φ contains only literals (positive or negative). We actually prove that satisfiability is in PTIME as soon as Φ only contains (ground) Horn clauses.
- A could be arbitrary, in principle, provided that it is consistent with π. In practice, we may assume that $A \cup \{\pi\}$ is a finite set of (possibly constrained) Horn clauses with equality (see [3] for a complete example). A typical example of an axiom (a consequence of IND-CCA, see [4]) is the *secrecy axiom*

$$\forall X, x, y. \quad \big[X; \mathsf{enc}(x, pk) \vdash n(y) \quad \rightarrow \quad X \vdash n(y)\big] \quad \| \quad sk \notin X$$

  The expression $n(y)$ represents a function that returns a random number. The formula states that the encryption of x does not help in deducing the nonce $n(y)$, unless the decryption key sk appears as a plain text of some term in X.

The problem that we consider in this paper is then the following one: when is such a satisfiability check tractable?

1.2 Difficulties

Following the approach of D. Mc Allester [10], D. Basin and H. Ganzinger [6] show that, if a set of Horn clauses is saturated, with respect to a well suited ordering and a well suited notion of redundancy, then the associated inference system is tractable. The main restriction in this paper is on the ordering with respect to which the clauses have to be saturated: given a ground term t, there should be only polynomially many terms smaller than t. (The subterm ordering, is an example. The term embedding does not satisfy this property).

However, the Horn clauses derived from security assumptions are beyond the scope of these results for several reasons that we describe below.

- The deducibility predicate $\vdash$ can be seen as a variadic predicate symbol, whose arguments (except the last one) are unordered. This is a problem, since Basin and Ganzinger's method yields an NP decision procedure with such a predicate: even if A is saturated (modulo the set axioms for the left part of the $\vdash$ predicate), when we use A to reduce a ground atom $S \vdash t$, potentially all subsets of S will be considered (see Section 3 for an example).
- Axioms (i.e. Horn clauses) are constrained. A priori, this is not an obstacle to the Basin and Ganzinger procedure, as the constraints can be checked on each superposition between an axiom and a ground clause. However, the very notion of saturation of a set of constrained clauses is an issue (as reported for instance in [12] for basic strategies or [8] for order constraints). In short: we cannot assume our set of axioms to be saturated.
- Clauses contain an equality predicate. This is not too tricky, since we may assume that A does not contain any equality. Hence equalities appear only as ground literals. We can then easily extend Basin and Ganzinger algorithm to clauses modulo a ground equational theory.

1.3 Overview of the Results and Proofs

Including a variadic predicate. We consider sets of ground Horn clauses with equality, whose atomic formulas may (also) be $S \vdash t$ where S is a finite set of (ground) terms and t is a ground term, together with a saturated set of clauses $\mathcal{A}$ with no deducibility predicate and the following set of clauses A_0:

$$A_0 = \begin{cases} X \vdash x \to X; y \vdash x & \text{weakening} \\ X \vdash x, \quad Y; x \vdash y \to X; Y \vdash y & \text{transitivity} \\ \to x \vdash x & \text{reflexivity} \\ X_1 \vdash x_1, \ldots, X_n \vdash x_n \to X_1; \ldots; X_n \vdash f(x_1, \ldots, x_n) & f \text{ function symbol} \end{cases}$$

Note that the left argument of $\vdash$ is a set. We write $X; x$ for $X \cup \{x\}$ and $X; Y$ for $X \cup Y$ and we compute modulo the set properties.

We prove first that satisfiability of such a set of clauses is in PTIME, therefore extending Basin and Ganzinger result, on the one hand with equalities (this is not the difficult part) and on the other hand with the deducibility predicate.

The main idea then is to use another layer of the ground Horn clauses entailment problem: given $S_1 \vdash t_1, \ldots, S_n \vdash t_n, S \vdash t$, whether $S_1 \vdash t_1, \ldots, S_n \vdash t_n$ entails $S \vdash t$ can be solved in PTIME. This is done by transforming literals $S \vdash t$ into clauses $S \to t$. Since the resulting clauses do not contain $\vdash$ anymore, this can be used as an oracle in a (modified) ground Horn clauses entailment problem.

Adding axioms on the deducibility predicate. The previous result is not sufficient for our purpose as, for instance, simple axioms such as secrecy (provided in Section 1.1) cannot be expressed in the considered fragment.

We therefore extend the previous results, adding formulas of the form

$$S \vdash x, \quad S; u(x) \vdash t(y) \quad \to \quad S \vdash t(y)$$

$$S; u(x) \vdash v(y) \quad \to \quad S \vdash v(y)$$

These formulas are relevant for our application. Indeed, the secrecy axiom described in Section 1.1 is an axiom of the second form. The axioms of the first form are useful to express e.g. non-malleability of encryption:

$$\forall X, x, y. \quad X \vdash x, \quad X; \mathsf{dec}(x, k) \vdash n(y) \quad \to \quad X \vdash n(y) \qquad \| \quad P(x)$$

The decryption of a deducible message x does not help to learn a nonce $n(y)$, provided that x does not appear as subterm of X, which can be encoded in a predicate P.

We show again in this case that the satisfiability is in PTIME. The first idea consists in seeing these clauses as new inference rules. For instance the first above axiom can be seen as a generalized cut (it is a cut when $u(x) = x$). As before, we first consider the entailment problem for deduction atomic formulas, which in turn can be seen as an entailment problem for Horn clauses. This can also be easily reduced to the problem of deducing the empty clause.

We design a complete strategy for this extended deduction system for which the proof search is in PTIME. Let us explain how it works. With the usual cut rule (and not the extended one above), whether the empty clause can be derived, can be decided in PTIME using a unit strategy. This is not the case with an extended cut rule. However, introducing some new rules and additional syntactic constructions, we design a proof system, whose expressive power is the same as the original proof system, and for which the unit strategy is complete, yielding a PTIME decision procedure. In other words, our strategy, that cannot be explained as a local strategy of application, can be reduced to a unit strategy, thanks to some memorization.

Adding constraints. Our application case requires to consider constraints, typically expressing that some term does not occur in the left side of a deduction relation. Such constraints have good stability properties: if they are satisfied by two sets of literals, then they are satisfied by their union and, if a constraint is satisfied by a set of literals S, then it is satisfied by any subset of S. Our main

restriction is however that there are only a fixed set of possible constraints. We show again that the satisfiability is in PTIME.

We cannot simply use the previous strategy, checking that constraints are satisfied whenever we need to apply them. The extended deduction system of the previous section is proved to be complete by a proof transformation that may not preserve constraint satisfaction. We therefore refine the strategy, memorizing additional information in the formulas: on the one hand, we store the constraints that are necessarily satisfied by all instances of the clause (this is inherited in the deduction rules) and, on the other hand, the constraints that have to be satisfied in the remainder of the proofs. Using this new syntax and inference rules, we show that they do not increase the expressiveness and yet that the unit strategy is refutation complete for these new rules. This shows the PTIME membership.

In the next step, we show that the entailment problem is decidable in PTIME in this new syntax. We need however to memorize a third component, which depends on the instance of the entailment problem.

Final result. From the previous paragraphs, we can build a PTIME entailment algorithm which, given $S_1 \vdash t_1 \dots S_n \vdash t_n, S \vdash t$ and clauses

$$A_1 = \begin{cases} S \vdash x, \quad S; u_i(x) \vdash t(y) \quad \rightarrow \quad S \vdash t(y) \quad \| \quad \Gamma_i \\ S; s_j(x) \vdash v(y) \quad \rightarrow \quad S \vdash v(y) \quad \| \quad \Delta_j \end{cases}$$

where Γ_i, Δ_j are finite sets of constraints, decides in PTIME whether $S_1 \vdash t_1, \dots, S_n \vdash t_n, A_1, A_0, \mathcal{A} \models S \vdash t$.

This algorithm can be used as an oracle in a variant of the Basin and Ganzinger algorithm, to decide the satisfiability of a set of clauses including formulas extending A_0, A_1 together with ground clauses with equality. Altogether, we obtain a PTIME procedure for arbitrary ground clauses and saturated Horn clauses (as in Basin & Ganzinger), together with the aforementioned clauses. This is exactly what we needed for our application, that is checking satisfiability of clauses corresponding to the computational security of a protocol.

Beyond our tractability results, we hope that our techniques and ideas of memorization can be reused in other contexts for the design of efficient strategies.

2 Formal Setting

Let $\mathcal{F}$ be a finite set of function symbols (together with their arity) and $\mathcal{P}$ be a finite set of predicate symbols together with their arity. $T(\mathcal{F})$ is the set of ground terms built on $\mathcal{F}$ (which is assumed to contain at least one constant) and $T(\mathcal{F}, \mathcal{X})$ is the set of terms built on $\mathcal{F}$ and a set of variable symbols $\mathcal{X}$. We also use set variables (written using upper case letters $X, Y, Z, ...$) ranging in a set $\mathcal{SX}$ and a function symbol, denoted by a semicolon, for set union. *Extended terms* $ET(\mathcal{F}, \mathcal{X}, \mathcal{SX})$ are expressions $s_1; \dots; s_n$ where $s_i \in T(\mathcal{F}, \mathcal{X}) \cup \mathcal{SX}$. As a shortcut, when $n = 0$ in the previous definition we denote the extended term as $\emptyset$. A *basic ordering* is an ordering on terms, which is : (1) Compatible with

substitutions and (2) such that, for every ground term t, the number of terms smaller than t is polynomial in the size of t. (An example of such an ordering is the subterm ordering).

Atomic formulas are of the following forms:

- $P(t_1, \ldots, t_n)$ where $P \in \mathcal{P}$ and $t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{X})$
- $t_1 = t_2$ where $t_1, t_2 \in T(\mathcal{F}, \mathcal{X})$
- $S \vdash t$ where $t \in T(\mathcal{F}, \mathcal{X})$ and $S \in ET(\mathcal{F}, \mathcal{X}, \mathcal{SX})$.

We consider clauses that are built on these atomic formulas. The axioms for the set theory ACIN (associativity, commutativity, idempotence and neutral element $\emptyset$) are implicitly assumed without mention on the left side of the $\vdash$. As usual, Horn clauses are clauses with at most one positive literal.

Given an extended term S and a substitution σ, mapping variables of $\mathcal{SX}$ to finite subsets of $T(\mathcal{F})$ and variables of $\mathcal{X}$ to terms in $T(\mathcal{F})$, $S\sigma$ is defined by $\emptyset\sigma = \emptyset$, $(s; S)\sigma = \{s\sigma\} \cup S\sigma$ if $s \in T(\mathcal{F}, \mathcal{X})$, and $(X; S)\sigma = X\sigma \cup S\sigma$ if $X \in \mathcal{SX}$.

3 Tractability of Deducibility Axioms

We first consider the consistency problem of a very specific case: let $\mathcal{C}$ be a set of ground clauses built on the deducibility predicate only. Is $\mathcal{C} \cup \{\rightarrow X; x \vdash x, \quad X \vdash x \rightarrow X; y \vdash x, \qquad X \vdash x,\ X; x \vdash y \rightarrow X \vdash y\}$ consistent? (We call respectively *r(eflexivity)*, *w(eakening)* and *t(ransitivity)* the three last clauses).

Consider for instance a ground clause $a_1, \ldots, a_n \vdash a \rightarrow \bot$. If we simply use a unit resolution strategy (which is refutation complete for Horn clauses), this single clause, together with the weakening clause, may generate all unit clauses $S \vdash a \rightarrow \bot$ where $S \subseteq \{a_1, \ldots, a_n\}$. This should be avoided since we seek for a polynomial time algorithm. Similar problems occur with transitivity, if we try to use binary resolution with a simple strategy. Here is a more concrete example.

Example 1. Let $\mathcal{C} = \{a_1; a_2; a_3 \vdash a_0 \rightarrow \bot, \quad \rightarrow a_1; a_4 \vdash a_0, \quad \rightarrow a_2 \vdash a_4\}$. $\mathcal{C} \cup \{w, t\}$ is provably unsatisfiable using binary resolution modulo ACIN only.

$$\dfrac{\dfrac{\rightarrow a_1; a_4 \vdash a_0 \quad X_1 \vdash x_1 \rightarrow X_1; y_1 \vdash x_1}{\rightarrow a_1; a_4; y_1 \vdash a_0} \qquad X_2 \vdash x_2,\ \ X_2; x_2 \vdash y_2 \rightarrow X_2 \vdash y_2}{a_1; y_1 \vdash a_4 \rightarrow a_1; y_1 \vdash a_0}$$

with unifiers $X_1 = a_1; a_4$, $X_2 = a_1; y_1$, $x_1 = a_0$, $x_2 = a_4$ and $y_2 = a_0$

$$\dfrac{a_1; y_1 \vdash a_4 \rightarrow a_1; y_1 \vdash a_0 \qquad \dfrac{\rightarrow a_2 \vdash a_4 \quad X_3 \vdash x_3 \rightarrow X; y_3 \vdash x_3}{\rightarrow a_2; y_3 \vdash a_4}}{\rightarrow a_1; a_2 \vdash a_0}$$

with unifiers $X_3 = a_2$, $y_1 = a_2$ and $y_3 = a_1$

and

$$\dfrac{\dfrac{\rightarrow a_1; a_2 \vdash a_0 \quad X_4 \vdash x_4 \rightarrow X_4; y_4 \vdash x_4}{\rightarrow a_1; a_2; y_4 \vdash a_0} \qquad a_1; a_2; a_3 \vdash a_0 \rightarrow \bot}{\bot}$$

with unifiers $X_4 = a_1; a_2$, $x_4 = a_0$ and $y_4 = a_3$.

This derivation introduces the clause $\rightarrow a_1; a_2 \vdash a_0$, where $a_1; a_2$ is a new set (i.e. it does not appear in the initial sets). This is actually unavoidable: any derivation of the empty clause requires as an intermediate step the derivation of either $\rightarrow a_1; a_2 \vdash a_0$ or $a_1; a_4; a_3 \vdash a_0 \rightarrow \bot$. Both of them involve sets that are not in the initial class.

However if we move from the object level to the meta-level, viewing weakening and transitivity as inference rules and deducibility atoms as clauses, we can at least solve this very particular case. More precisely, consider the inference system:

$$\frac{}{X; x \vdash x}\, R \qquad \frac{X \vdash x}{X; y \vdash x}\, W \qquad \frac{X \vdash x \quad X; x \vdash y}{X \vdash y}\, T$$

where X is a logical variable ranging over extended terms and x, y are logical variables ranging over terms.
Let $\Vdash_{R,W,T}$ be the derivability relation associated with these two inference rules.

Lemma 2. *Given ground atomic formulas $S_1 \vdash t_1, \ldots, S_n \vdash t_n$ and $S \vdash t$, we can decide in linear time whether $\{S_1 \vdash t_1, \ldots, S_n \vdash t_n\} \Vdash_{R,W,T} S \vdash t$.*

Proof. We associate with each term occurring in $S_1 \cup \ldots \cup S_n \cup S \cup \{t_1, \ldots, t_n, t\}$ a proposition variable. We claim that $S_1 \vdash t_1, \ldots, S_n \vdash t_n \Vdash_{R,W,T} S \vdash t$ iff $S \rightarrow t$ is derivable from $S_1 \rightarrow t_1, \ldots, S_n \rightarrow t_n$ using the propositional binary resolution, excluded middle and weakening rules only. Indeed we notice that T, R and W can be simulated by resolution and excluded middle. For W the proof rewriting is straightforward. We present the proof rewriting for T and R (the double bar stands for multiple applications of a rule) :

$$\frac{S \vdash t \quad S; t \vdash u}{S \vdash u}\, T \quad \Longrightarrow \quad \frac{S \rightarrow t \quad S, t \rightarrow u}{S \rightarrow u}\, Res$$

$$\frac{}{S; t \vdash t}\, R \Longrightarrow \frac{\dfrac{}{t \rightarrow t}\, Excl}{S, t \rightarrow t}\, Weak$$

Conversely the resolution, excluded middle and weakening can be simulated by R, T and W. The proof rewriting is straightforward for excluded middle and weakening, we only present it for resolution :

$$\frac{S_1 \rightarrow t \quad S_2, t \rightarrow u}{S_1, S_2 \rightarrow u}\, Res \quad \Longrightarrow \quad \frac{\dfrac{S_1 \vdash t}{S_1; S_2 \vdash t}\, W \qquad \dfrac{S_2; t \vdash u}{S_1; S_2; t \vdash u}\, W}{S_1; S_2 \vdash u}\, T$$

With these observations we now have that derivability of $S \rightarrow t$ is equivalent to unsatisfiability of $S_1 \rightarrow t_1, \ldots, S_n \rightarrow t_n, S, \neg t$ (where S_i is a shortcut for the conjunction of propositional variables corresponding to terms occurring in S_i), which can be decided in linear time: it is a HornSat problem.

Now, the trick of viewing the clauses w, t as new inference rules allows to decide our problem in PTIME. We write $\Vdash_{Res_u+R+W+T}$ for the derivability with inference rules R, W, T and unit resolution.

Lemma 3. *Given a set of ground Horn clauses (built on $\vdash$) $\mathcal{C}$, the satisfiability of $\mathcal{C} \cup \{r, w, t\}$ is decidable in cubic time.*

Proof. We show first that $\mathcal{C} \cup \{r, w, t\}$ is unsatisfiable iff the empty clause can be derived from $\mathcal{C}$, using unit resolution $R + W + T$. If we can derive the empty clause in this system, then we can derive the empty clause from $\mathcal{C} \cup \{r, w, t\}$ by resolution, thanks to simple proof rewriting rules :

$$\frac{}{S; t \vdash t}\, R \quad \Longrightarrow \quad S; t \vdash t \text{ (instance of } r)$$

$$\frac{\overset{\pi_1}{S \vdash t}}{S; u \vdash t}\, W \quad \Longrightarrow \quad \frac{\overset{\pi_1}{S \vdash t} \qquad X \vdash x \;\rightarrow\; X; y \vdash x}{S; u \vdash t}\, Res$$

$$\frac{\overset{\pi_1}{S \vdash t} \qquad \overset{\pi_2}{S; t \vdash u}}{S \vdash u}\, T \quad \Longrightarrow \quad \frac{\dfrac{\overset{\pi_1}{S \vdash t} \qquad X; x \vdash y,\; X \vdash x \;\rightarrow\; X \vdash y}{S; t \vdash y \rightarrow S \vdash y}\, Res \qquad \overset{\pi_2}{S; t \vdash u}}{S \vdash u}\, Res$$

Conversely, if we cannot derive the empty clause from $\mathcal{C}$ using unit resolution $R + W + T$, then let $\mathcal{M} = \{S \vdash u \mid \mathcal{C} \Vdash_{Res_u+R+W+T} S \vdash u\}$. We claim that $\mathcal{M}$ is a model of $\mathcal{C} \cup \{r, w, t\}$: As $\mathcal{M}$ is closed by R, W, T, it is a model of $\{r, w, t\}$ and, if $B_1, \ldots, B_n \rightarrow H \in \mathcal{C}$, then either $B_i \notin \mathcal{M}$ for some i or else, by construction, for every i, $\mathcal{C} \Vdash_{Res_u+R+W+T} B_i$, hence, by unit resolution, $\mathcal{C} \Vdash_{Res_u+R+W+T} H$. In all cases, $\mathcal{M} \models B_1, \ldots, B_n \rightarrow H$.

It only remains to prove that whether $\mathcal{C} \Vdash_{Res_u+R+W+T} \bot$ or not can be decided in cubic time. Let $\mathcal{B}$ be the set of atomic formulas occurring in $\mathcal{C}$. Let $\mathcal{M}$ be the least fixed point of

$$f(X) = \{S \vdash u \in \mathcal{B} \mid \mathcal{C} \cup X \Vdash_{Res_u} S \vdash u \text{ or } \mathcal{C} \cup X \Vdash_{R+W+T} S \vdash u\}$$

Since f is monotone, there is a least fixed point, which is contained in $\mathcal{B}$. Computing $\mathcal{M}$ can be performed in cubic time, as there are at most $|\mathcal{B}|$ iterations and each step requires at most a linear time, thanks to Lemma 2.

If the empty clause can be derived from $\mathcal{M}, \mathcal{C}$ using unit resolution, then $\mathcal{C} \Vdash_{Res_u+R+W+T} \bot$. Let us show the converse implication. For this, we prove, by induction on the proof size that, for every atomic formula $S \vdash t \in \mathcal{B}$, $\mathcal{C} \Vdash_{Res_u+R+W+T} S \vdash t$ implies $S \vdash t \in \mathcal{M}$.

If the last rule of the proof is a unit resolution, then the proof can be written:

$$\dfrac{\pi_1}{S_1 \vdash t_1} \quad \dfrac{\dfrac{\pi_2}{S_2 \vdash t_2} \quad \dfrac{\dfrac{\pi_n}{S_n \vdash t_n} \quad \dfrac{}{S_1 \vdash t_1, \ldots, S_n \vdash t_n \to S \vdash t}\,(S_1 \vdash t_1, \ldots, S_n \vdash t_n \to S \vdash t) \in \mathcal{C}}{S_1 \vdash t_1, \ldots, S_{n-1} \vdash t_{n-1} \to S \vdash t}}{\vdots}}{S_1 \vdash t_1, S_2 \vdash t_2 \to S \vdash t}}{S_1 \vdash t_1 \to S \vdash t}}{S \vdash t}$$

$S_1 \vdash t_1, \ldots, S_n \vdash t_n \in \mathcal{B}$ and, by induction hypothesis, $S_1 \vdash t_1, \ldots, S_n \vdash t_n \in \mathcal{M}$. It follows that $\mathcal{M}, \mathcal{C} \Vdash_{Res_u} S \vdash t$, hence $S \vdash t \in f(\mathcal{M}) = \mathcal{M}$.

If the last rule of the proof is W or T, then there are atomic formulas $S_1 \vdash t_1, \ldots, S_n \vdash t_n$ such that $S_1 \vdash t_1, \ldots, S_n \vdash t_n \Vdash_{R+W+T} S \vdash t$ and, for every i, either $S_i \vdash t_i \in \mathcal{C}$ or the last rule in the proof of $S_i \vdash t_i$ is a resolution step and, as noticed previously all, $S_i \vdash t_i$ are in $\mathcal{B}$. In all cases $S_i \vdash t_i \in \mathcal{B}$ and, by induction hypothesis, $S_i \vdash t_i \in \mathcal{M}$. By definition of the function f, $S \vdash t \in f(\mathcal{M}) = \mathcal{M}$.

If $\mathcal{C} \Vdash_{Res_u+R+W+T} \bot$, then there is a negative clause $S_1 \vdash t_1, \ldots, S_n \vdash t_n \to \bot$ in $\mathcal{C}$ such that, for every i, $\mathcal{C} \Vdash_{Res_u+R+T+W} S_i \vdash t_i$, hence $S_i \vdash t_i \in \mathcal{M}$ as we just saw. Then $\bot$ can be deduced from $\mathcal{C}, \mathcal{M}$ using unit resolution (which can be decided in linear time again).

Example 4. Applying Lemma 3 to Example 1, checking the satisfiability of $\mathcal{C} \cup \{r, w, t\}$ simply amounts into checking whether $\{a_1; a_4 \to a_0, \quad a_2 \to a_4\}$ (does not) entail $a_1; a_2; a_3 \to a_0$.

3.1 Adding Equality

Now, we assume that atomic formulas in $\mathcal{C}$ may contain equalities on terms (not extended terms). The equality axioms (the equality is a congruence) are implicit in what follows.

Lemma 5. *Given a set of ground Horn clauses (built on $\vdash$ and $=$) $\mathcal{C}$, the satisfiability of $\mathcal{C} \cup \{r, w, t\}$ is decidable in polynomial time.*

Proof sketch: First, we extend Lemma 2. Given a finite set of equations E, the transitivity rule is extended to

$$\dfrac{x =_E z \quad X \vdash x \quad X; z \vdash y}{X \vdash y}\, T(E)$$

Given $S_1 \vdash t_1 \ldots, S_n \vdash t_n, S \vdash t$ and a finite set of ground equations E, we can decide in polynomial time whether $S_1 \vdash t_1, \ldots, S_n \vdash t_n \Vdash_{R,W,T(E)} S \vdash t$. We only have to check, for every pair of terms u, v in $S_1, t_1, \ldots, S_n, t_n, S, t$, whether $u =_E v$. This can be completed in polynomial time, for instance using

a quadratic time congruence closure algorithm. We may then choose one representative for each congruence class and use the same proof as in Lemma 2 on the representatives.

Then, as in Lemma 3, we consider the set $\mathcal{B}_\vdash$ of atomic formulas $S \vdash t$ occurring in $\mathcal{C}$ and $\mathcal{B}_=$ the set of equations occurring as atomic formulas in $\mathcal{C}$. We consider the monotone function

$$f(X,E) = (\ \{S \vdash t \in \mathcal{B}_\vdash \mid \mathcal{C} \cup X \Vdash_{Res_u(E)} S \vdash t \text{ or } \mathcal{C} \cup X \Vdash_{R+W+T(E)} S \vdash t\},\\ \{s = t \in \mathcal{B}_= \mid \mathcal{C} \cup X \Vdash_{Res_u(E)} s = t\}\ )$$

where $\Vdash_{Res_u(E)}$ is the unit resolution on representatives of the clauses w.r.t. E.

The least fixed point of f can be computed in polynomial time, as each iteration is polynomial and there is a polynomial number of iterations. $\mathcal{C} \cup \{r, w, t\}$ is satisfiable iff the empty clause can not be derived by unit resolution from this least fixed point.

3.2 Adding a Function Axiom

We extend now the clauses specifying $\vdash$ with the clauses (denoted by $f(\mathcal{F})$ later): $X \vdash x_1, \quad \cdots \quad X \vdash x_n \quad \rightarrow \quad X \vdash g(x_1, \ldots, x_n)$, for every function symbol g in a set of function symbols $\mathcal{F}$ (which is later omitted).

Lemma 6. *Given a set of ground Horn clauses (built on $\vdash$ and $=$) $\mathcal{C}$, the satisfiability of $\mathcal{C} \cup \{r, w, t\} \cup f(\mathcal{F})\}$ is decidable in polynomial time.*

Proof sketch: Again, adding an inference F_g for each of the new clauses, we first show that deciding $S_1 \vdash t_1, \ldots, S_n \vdash t_n \Vdash_{R+W+T(E)+\{F_g, g \in \mathcal{F}\}} S \vdash t$ is in PTIME. We use a proof similar to Lemma 5, with an additional observation: given a finite set E of ground equations and ground terms $t_1, \ldots, t_n, t$, we can decide in PTIME whether there is a context C (built using function symbols in $\mathcal{F}$) such that $C[t_1, \ldots, t_n] =_E t$. To prove this we may for instance compute a tree automaton $\mathcal{A}_t$ that recognizes the equivalence class of t and decide the emptiness of the intersection of $L(\mathcal{A}_t)$ with the set of terms $C[t_1, \ldots, t_n]$. All these steps can be performed in a total time, which is polynomial in the size of E, $t_1, \ldots, t_n, t$.

Example 7. $b \vdash c, \vdash a \Vdash_{R+W+T(g(g(a))=b)+F_g} \vdash c$ since there is a context C (with $C[\_] = g(g(\_))$) such that $C[a] = b$.

4 More Clauses Using the Deducibility Predicate

We now enrich the class of clauses involving the deducibility predicate. Given a term p (later called the *pattern*), we consider a finite set of clauses of the following forms:

$c_s(u)$: $X; u \vdash p \rightarrow X \vdash p$ where u is a term that does not share variables with p
$c_c(w)$: $X \vdash y, X; w \vdash p \rightarrow X \vdash p$ where w is a term that does not share variables with p, and y is a variable of w.

Example 8. The secrecy axiom described in introduction

$$X; \mathsf{enc}(x, pk) \vdash n(y) \quad \rightarrow \quad X \vdash n(y)$$

is an instance of the first class of clauses above, with $p = n(y)$ and $u = \mathsf{enc}(x, pk)$. The condition $sk \notin X$ requires constraints, that are considered in Section 5.

As explained in the previous section, we may turn the additional clauses into new inference rules, using $\leq_E$, the matching modulo E (a term t satisfies $u \leq_E t$ if there is a substitution σ such that $t =_E u\sigma$).

$$\frac{u \leq_E x \quad X; x \vdash p}{X \vdash p}\,\mathsf{Str}_u \qquad \frac{(y, w) \leq_E (x, z) \quad X \vdash x \quad X; z \vdash p}{X \vdash p}\,\mathsf{Cut}_w$$

Let $\mathcal{I}$ be the inference system defined by a finite collection of rules $\mathsf{Str}_u, \mathsf{Cut}_w$, the rules $R, W, T(E)$ for a finite set of ground equations E and the rules F_g for a set of function symbols g.

We are going to prove that, again, $\mathcal{I}$ can be decided in polynomial time. However, we cannot use the same proof as in the previous section. $S_1 \vdash t_1, \ldots, S_n \vdash t_n \Vdash_{\mathcal{I}} S \vdash t$ can no longer be reduced to a problem $S_1 \rightarrow t_1, \ldots, S_n \rightarrow t_1, S \Vdash_{Res_u} t$ (modulo a PTIME oracle).

Example 9. Assume E is empty and we have a single rule $\mathsf{Cut}_{f(x,k)}$ for the pattern $p = n$. $f(a, k) \vdash f(b, k),\ f(b, k) \vdash n \Vdash_{\mathcal{I}} a \vdash n$:

$$\dfrac{\dfrac{}{a \vdash a}\,R \qquad \dfrac{\dfrac{f(a,k) \vdash f(b,k)}{a; f(a,k) \vdash f(b,k)}\,W \qquad \dfrac{f(b,k) \vdash n}{a; f(a,k); f(b,k) \vdash n}\,W}{a; f(a,k) \vdash n}\,T}{a \vdash n}\,\mathsf{Cut}_{f(x,k)}$$

We cannot use a unit version of T (or resolution) in this example. And moving to a general binary resolution would yield an exponential procedure.

As before, after turning the clauses into inference rules, we turn the deducibility atomic formulas into clauses. We call again $\mathcal{I}$ the resulting inference system. We have to be careful however: this is a purely syntactic transformation and the inference rules resulting from this translation are no longer correct in a classical semantics. For instance Cut_w becomes

$$\frac{A_1, \ldots, A_n \rightarrow y \qquad w, B_1, \ldots, B_m \rightarrow p}{A_1, \ldots, A_n, B_1, \ldots, B_m \rightarrow p}$$

where the premises are matched modulo a set of ground equations E.

In order to apply a simple fixed point computation, we would like to be able to transform any proof into a unit strategy proof. Since this is not possible with the current proof system (as shown by Example 9), we introduce additional inference rules that will allow such a strategy, however bookkeeping what the rest of the proof owes, in order to enable a translation back into the original proof system.

Example 10. Continuing Example 9, the unit proof of $\to n$ from the hypotheses $\to a,\ f(a,k) \to f(b,k),\ f(b,k) \to n$ will look like this:

$$\dfrac{\dfrac{\to a \qquad f(a,k) \to f(b,k)}{\to_p f(b,k)}\,\mathsf{Cut}^1_{f(x,k)} \qquad f(b,k) \to n}{\to n}\,\mathsf{Cut}^2$$

The rule Cut^1_u is a generalisation of Cut_u since the constraint of being an instance of the pattern p on the right is dropped. It bookkeeps however a duty as a mark p on the arrow. The mark on a clause $S \to_p t$ can in turn be erased only when a clause $S', t \to p$ is one of the premises. Such a mechanism allows both to use a complete unit strategy and to enable reconstructing an original proof from the extended one, as we will prove (here the annotation is erased in the last rule as the second premise is an instance of $S, f(x,k) \vdash n$).

Intuitively, the head s of a marked clause can only be used in a proof that will end up deriving an instance of the pattern.

We extend the syntax, allowing both unmarked clauses $S \to t$ and marked clauses $S \to_p t$. For simplicity, we first do not consider the set of ground equations E nor the function axioms. We write $S \to_? t$ when it does not matter whether the arrow is marked or not. We then consider the inference system $\mathcal{J}$ consisting of $T(E)$, W and the following rules (for each Cut_w there are two rules Cut^i_w and for each rule Str_u there are two rules Str^i_u):

$$\dfrac{A_1, \ldots, A_n \to_? x \qquad B_1, \ldots, B_m, w \to_? v}{A_1, \ldots, A_n, B_1, \ldots, B_m \to_p v}\,\mathsf{Cut}^1_w$$

$$\dfrac{A_1, \ldots, A_n \to_? x \qquad w, B_1, \ldots, B_m \to p}{A_1, \ldots, A_n, B_1, \ldots, B_m \to p}\,\mathsf{Cut}^2_w$$

$$\dfrac{A_1, \ldots, A_n \to_? x \qquad B_1, \ldots, B_m, x \to_? v}{A_1, \ldots, A_n, B_1, \ldots, B_m \to_? v}\,\mathsf{Cut}^1$$

in which the conclusion is marked iff one of the premises is marked.

$$\dfrac{A_1, \ldots, A_n \to_? x \qquad x, B_1, \ldots, B_m \to p}{A_1, \ldots, A_n, B_1, \ldots, B_m \to p}\,\mathsf{Cut}^2$$

$$\dfrac{A_1, \ldots, A_n, u \to_? x}{A_1, \ldots, A_n \to_p x}\,\mathsf{Str}^1_u \qquad \dfrac{A_1, \ldots, A_n, u \to_? p}{A_1, \ldots, A_n \to p}\,\mathsf{Str}^2_u$$

Note that the above system has no classical semantics.

Lemma 11. *Let $\mathcal{S}$ be a set of ground clauses, and s be a ground term. In case $E = \emptyset$ and removing the function and reflexivity axioms from $\mathcal{I}$, $\mathcal{S} \Vdash_{\mathcal{I}} \to s$ if and only if $\mathcal{S} \Vdash_{\mathcal{J}} \to s$.*

Proof sketch: For one implication we prove that W is not necessary, hence $\mathcal{I}$ can be simulated by $\mathcal{J}$. For the other implication, we rewrite a proof in $\mathcal{J}$ as follows. We consider a last rule that introduces a mark. Since the marks must eventually disappear, there is also a matching rule that removes the mark. This part of proof is then rewritten as explained on the following example:

$$\dfrac{\dfrac{\dfrac{\dfrac{S_1 \to t_1 \quad S, w\sigma \to v\sigma}{S_2 \to t_2 \qquad S_1, S \to_p v\sigma}\,\mathsf{Cut}^1_w}{S_2' \to_p v\sigma}\,\mathsf{Cut}^1_{w_2}}{\vdots}}{\dfrac{S_n \to t_n \qquad\qquad S_n' \to_p v\sigma}{S_0, S_n' \to p\theta}\,\mathsf{Cut}^1_{w_n}}\qquad \dfrac{}{S_0, t\sigma' \to p\theta}\,\mathsf{Cut}^2_t$$

rewrites to

$$\dfrac{\dfrac{\dfrac{\dfrac{S, w\sigma \to v\sigma \quad S_0, t\sigma' \to p\theta}{S_1 \to t_1 \qquad S_0, S, w\sigma \to p\theta}\,\mathsf{Cut}_t}{S_0, S_1, S \to p\theta}\,\mathsf{Cut}_w}{\vdots}}{\dfrac{S_n \to t_n}{S_0, S_n' \to p\theta}\,\mathsf{Cut}_{w_n}}$$

The proof rewriting terminates and we end up with a proof in $\mathcal{I}$. See Appendix A for more details.

The *unit* strategy for $\mathcal{J}$ consists in applying the rules only when $n = 0$ for the Cut^i_w rules (i.e. when the left premise of a Cut^i_w is a unit clause).

Lemma 12. *If $\mathcal{S} \Vdash_{\mathcal{J}} \to s$ then $\to s$ is derivable from $\mathcal{S}$ in $\mathcal{J}$ using the unit strategy.*

Proof sketch: We prove it by induction on the proof size. We assume w.l.o.g. that all proofs of literals (whether marked or not) labeling a node in the proof (except the root) use a unit strategy. We consider the last step that does not comply with the unit strategy. If $A_1, \ldots, A_n \to_? s$ is its conclusion, then all atoms $A_1, \ldots, A_n$ can be proved in $\mathcal{J}$ with the unit strategy. We therefore simplify the premises accordingly, which yields an inference rule complying with the unit strategy.

Theorem 13. *If $\mathcal{S}$ is a set of ground clauses built on $\vdash$, we can decide in PTIME the satisfiability of $\mathcal{S}$, together with T, W and finitely many clauses c_s, c_c, that are built on the same pattern p.*

Proof sketch: we first observe that, thanks to the lemmas 11 and 12 (and using a fixed point computation), given the ground atoms $S_1 \vdash t_1, \ldots, S_n \vdash t_n, S \vdash t$, it is possible to decide in PTIME whether $S_1 \vdash t_1, \ldots, S_n \vdash t_n \Vdash_{\mathcal{I}} S \vdash t$. We then conclude using an argument similar to the one given in Section 3.

4.1 Adding Other Predicate Symbols

We now consider the case where the clauses c_s, c_n, c_c are guarded with literals built on a set of predicate symbols $\mathcal{P}$ not containing $\vdash$ and that are defined using a saturated set of Horn clauses $\mathcal{A}_0$. For instance, $c_c(w)$ is extended to clauses of the form $P_1(s_1), \ldots, P_n(s_n),\ X \vdash y,\ X; w \vdash p\ \rightarrow X \vdash p$. The variables of $s_1, \ldots, s_n$ are assumed to be a subset of the variables of w, y.

We modify the rules Cut^i_w adding as premises the literals $P_1(s_1), ..., P_n(s_n)$. Lemma 11 still holds, provided we add to $\mathcal{S}$ finitely many ground atoms on the new alphabet of predicates. To see this, we need to check that the proof transformation yields the same instances of $P_i(s_i)$. Lemma 12 is unchanged. These properties rely on the fact that guards (and their instances) do neither depend on the set variable X (nor its instances) nor on the instances of the pattern.

Theorem 13 can then be extended to this case: when computing the fixed point, the instances of applicable inference rules are known at each step and we only have to check whether the corresponding instances of the guards are consequences of $\mathcal{A}_0$ (and possibly a finite set of ground atoms), which can be performed in PTIME, thanks to [6]. As a consequence, we get:

Theorem 14. *Let $\mathcal{P}$ be a set of predicate symbols, not containing $\vdash, =$ and $\mathcal{A}_0$ be a set of Horn clauses built on $\mathcal{P}$ and which is saturated w.r.t. a basic ordering. If $\mathcal{S}$ is a set of ground clauses built on $\vdash$ (possibly with guards using $\mathcal{P}$), we can decide in PTIME the satisfiability of $\mathcal{S} \cup \mathcal{A}_0$, together with T, W and finitely many clauses c_n, c_s, c_c, that are built on the same pattern p and which may be guarded by atomic formulas that use the predicate symbols in $\mathcal{P}$.*

4.2 Adding Equality

We can extend again Theorem 14 to ground equalities in the atomic formulas of $\mathcal{S}$. The procedure is the same as in Lemma 5: for a fixed E, Lemmas 11 and 12 can be extended, considering representatives modulo $=_E$. Then we only have to compute a fixed point of a function f on the atomic formulas of $\mathcal{S}$, using the PTIME oracles provided by (extensions of) Lemmas 11 and 12.

5 The General Case

Finally, we extend the results of the previous section to clauses with constraints.

A *constraint* Γ is a formula interpreted as a subset of $((T(\mathcal{F}))^*)^n$ (n-tuples of finite sets of ground terms) if n is the number of free variables of Γ. We write $S_1, \ldots, S_n \models \Gamma$ when $(S_1, \ldots, S_n)$ belongs to this interpretation. A *constrained clause* is a pair of a clause and a constraint, which is written $\phi \quad \| \quad \Gamma$. Given a constrained clause $\phi \quad \| \quad \Gamma$, we let $[\![\phi \| \Gamma]\!] = \{\phi\sigma | \sigma \text{ satisfies } \Gamma\}$. A model of $\phi \quad \| \quad \Gamma$ is, by definition, a model of $[\![\phi \| \Gamma]\!]$. A constraint Γ is *monotone* if

- if $S_1, \ldots, S_n \models \Gamma$ and, for every i, $S'_i \subseteq S_i$, then $S'_1, \ldots, S'_n \models \Gamma$
- if $S_1, \ldots, S_n \models \Gamma$ and $S'_1, \ldots, S'_n \models \Gamma$, then $S_1 \cup S'_1, \ldots, S_n \cup S'_n \models \Gamma$.

We typically use constraints of the form $t \notin X$ (where $t \in T(\mathcal{F})$), satisfied by any S that does not contain t as subterm. Such constraints are monotone.

Adding a fixed set of possible constraints increases significantly the difficulty: Lemmas 11 and 12 no longer hold, as shown by the following example:

Example 15. Consider the clause $c_{f(y,k)} : \quad X \vdash y, \;\; X; f(y,k) \vdash n \;\; \rightarrow \;\; X \vdash n \quad \| f(a,k), f(b,k), f(c,k) \notin X$. Consider the ground deducibility formulas: $\mathcal{S} = \{(f(a,k) \vdash f(b,k), \;\; f(b,k); f(c,k) \vdash n\}$. Does $c_{f(y,k)}$ and $\mathcal{S}$ entail $a; c \vdash n$?

Following the procedure of Section 4,

$$\dfrac{\dfrac{\dfrac{\rightarrow a \quad f(a,k) \rightarrow f(b,k)}{\rightarrow_p f(b,k)}\,\mathsf{Cut}^1_{f(y,k)} \qquad f(b,k); f(c,k) \rightarrow n}{f(c,k) \rightarrow n}\,\mathsf{Cut}^2 \qquad \rightarrow c}{\rightarrow n}\,\mathsf{Cut}^2_{f(y,k)}$$

in which each $\mathsf{Cut}^i_{f(y,k)}$ satisfies the constraint that $f(a,k), f(b,k), f(c,k)$ do not appear in the context: the instance of X is empty in each case. The procedure would then incorrectly answers "yes" to the entailment question.

Indeed, the proof rewriting of Lemma 11 yields the following (invalid) proof, in which the constraints are *not* satisfied in the first application of $\mathsf{Cut}_{f(x,k)}$, since the corresponding instance of X is the one element set $f(c,k)$:

$$\dfrac{\dfrac{\dfrac{f(a,k) \rightarrow f(b,k) \;\; f(b,k); f(c,k) \rightarrow n}{f(a,k); f(c,k) \rightarrow n}\,Res \qquad \rightarrow a}{f(c,k) \rightarrow n}\,\mathsf{Cut}_{f(x,k)} \qquad \rightarrow c}{\rightarrow n}\,\mathsf{Cut}_{f(x,k)}$$

Our solution consists in designing another inference system, along the same ideas as before, for which Lemmas 11 and 12 still hold. To do so, we memorize more information in the mark (typically the constraints that need to be satisfied) so that the matching rule (removing the mark) can be applied only if the actual clauses would satisfy the constraints recorded in the mark.

Example 16. To explain the main idea, we give a simplified example of how the new proof system works. Coming back to Example 15, in our system we get:

$$\dfrac{\rightarrow a \quad f(a,k) \rightarrow f(b,k)}{\rightarrow_{f(a,b), f(b,k), f(c,k) \notin X} f(b,k)}\,\mathsf{Cut}^1_{f(y,k)}$$

But we cannot apply Cut^2 since its application requires that the context satisfies the constraint in the mark, which is not the case. We could apply a Cut^1, without removing the mark but then the mark could not be removed any more since the marks can never be removed from the "pattern premisse" of a Cut^i_w rule.

If the clause is less constrained, for instance assume that we only impose $f(b,k) \notin X$, then we can prove $\to n$ as follows:

$$\dfrac{\dfrac{\to c \qquad \dfrac{\dfrac{\to a \quad f(a,k) \to f(b,k)}{\to_{f(b,k)\notin X} f(b,k)}\,\mathsf{Cut}^1_{f(y,k)} \qquad f(b,k); f(c,k) \to n}{f(c,k) \to n}\,\mathsf{Cut}^2}{\to n}\,\mathsf{Cut}^2_{f(y,k)}$$

This time, we may remove the mark, as the instance of X is the singleton $\{f(c,k)\}$, that does not contain $f(b,k)$.

We get an analog of Lemmas 11 and 12, which yields a PTIME decision procedure (because the number of possible marks is fixed).

Theorem 17. *If $\mathcal{S}$ is a set of ground clauses built on $\vdash$, we can decide in PTIME the satisfiability of $\mathcal{S}$ together with T, W and finitely many constrained clauses c_s, c_c built on the same pattern p, provided the constraints are monotone.*

Again, this can be extended, as in the theorem 14, guarding the clauses with predicates that are defined by a saturated set of Horn clauses $\mathcal{A}_0$ (w.r.t. a basic ordering). This can be extended also to the case where $\mathcal{S}$ contains equalities.

6 Conclusion

We designed a technique for proving tractability of a collection of proof systems (or Horn clauses): the idea is to extend the proof system with marked clauses such that the expressivity is unchanged while the unit strategy becomes complete. Our technique captures a class of clauses relevant to a computer security application.

PTIME membership is obtained by nesting PTIME oracles. We did not succeed however in showing a more abstract combination result allowing, say, to combine two tractable inference systems, one of which depends on the other. For instance, when we add guards to another system (resp. equalities in the input clauses) we would like to get automatically a tractability property from the tractability of the system without guards (resp. without equality) and the tractability of the guards entailment (resp. tractability of the word problem).

Another perspective is to provide a more abstract statement of the proof method, which does not rely on the specific deducibility predicate. Moreover, our work is not fully complete since we did not consider the function and reflexivity axioms in the two last sections. We could also investigate the case of several patterns and/or constraints that involve both a (non-ground) term and a set.

References

1. Abadi, M., Rogaway, P.: Reconciling two views of cryptography (the computational soundness of formal encryption). J. Cryptology 15(2), 103–127 (2002)
2. Backes, M., Pfitzmann, B.: Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In: 17th IEEE Computer Science Foundations Workshop (CSFW 2004), pp. 204–218 (2004)

3. Bana, G., Adao, P., Sakurada, H.: Computationally complete symbolic attacker in action. In: 32nd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012), pp. 546–560 (2012)
4. Bana, G., Comon-Lundh, H.: Towards unconditional soundness: Computationally complete symbolic attacker. In: Degano, P., Guttman, J.D. (eds.) POST 2012. LNCS, vol. 7215, pp. 189–208. Springer, Heidelberg (2012)
5. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)
6. Basin, D., Ganzinger, H.: Automated complexity analysis based on ordered resolution. J. of the Association of Computing Machinery 48(1), 70–109 (2001)
7. Blanchet, B.: A computationally sound mechanized prover for security protocols. In: IEEE Symposium on Security and Privacy (S&P 2006), pp. 140–154 (2006)
8. Comon, H., Treinen, R.: The first-order theory of lexicographic path orderings is undecidable. Theoretical Computer Science 176(1-2), 67–87 (1997)
9. Datta, A., Derek, A., Mitchell, J.C., Warinschi, B.: Computationally sound compositional logic for key exchange protocols. In: 19th IEEE Computer Security Foundations Workshop (CSF 2006), pp. 321–334 (2006)
10. McAllester, D.: Automatic recognition of tractability in inference relations. Journal of the ACM 40(2) (1993)
11. Micciancio, D., Warinschi, B.: Soundness of formal encryption in the presence of active adversaries. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 133–151. Springer, Heidelberg (2004)
12. Nieuwenhuis, R., Rubio, A.: Handbook of Automated Reasoning, chapter Paramodulation-Based Theorem Proving. Elsevier Science and MIT Press (2001)

A Proof of Lemma 11

Lemma 11. *Let $\mathcal{S}$ be a set of ground clauses, and s be a ground term. In case $E = \emptyset$ and removing the function and reflexivity axioms from $\mathcal{I}$, $\mathcal{S} \Vdash_{\mathcal{I}} \rightarrow s$ if and only if $\mathcal{S} \Vdash_{\mathcal{J}} \rightarrow s$.*

Proof. We first prove that, if there is a proof Π of s in $\mathcal{I}$ from $\mathcal{S}$, then there is a proof Π' without W. Indeed, we may push W to the bottom of the proof as follows:

$$\dfrac{\dfrac{A_1, \ldots, A_n \rightarrow x}{A_1, \ldots, A_n, C \rightarrow x}\, W \qquad B_1, \ldots, B_m, w \rightarrow p}{A_1, \ldots, A_n, B_1, \ldots, B_m, C \rightarrow p}\, \mathsf{Cut}_w$$

can be rewritten to

$$\dfrac{\dfrac{A_1, \ldots, A_n, \rightarrow x \quad B_1, \ldots, B_m, w \rightarrow p}{A_1, \ldots, A_n, B_1, \ldots, B_m, \rightarrow p}\, \mathsf{Cut}_w}{A_1, \ldots, A_n, B_1, \ldots, B_m, C \rightarrow p}\, W$$

W also commutes with the rules Str_u. Since the proof of a unit clause cannot end with W, Π does not contain W.

Now let us show that if there is a proof of $\to s$ in $\mathcal{J}$ then there is a proof of $\to s$ in $\mathcal{I}$: Consider a minimal (in number of Cut^1, Cut^1_w, Str^1_u rules) proof Π of $S \to t$ in $\mathcal{J}$. Consider a subproof Π' of Π that uses once Cut^2_w, as a last inference rule. We show that Π' can be rewritten into a strictly smaller proof (w.r.t. the size). This contradicts the minimality of Π, hence this proves that the minimal size proof does not make use of any extra rule.

First note that, according to labels inheritance, once a clause is annotated, then the label cannot be removed completely, unless we apply Cut^2_w or Cut^2. Since the leaves of Π' are not annotated, we can write Π' as :

$$\dfrac{\dfrac{\dfrac{\vdots\ \pi_1}{S^1 \to t}\, R^1 \qquad \vdots}{S^n \to_p t}\, R^n \qquad \overset{\pi_2}{S, w\sigma \to p\sigma}}{S^n, S \to p\sigma}\, \mathsf{Cut}^2_w$$

where $R^1, \ldots, R^n$ are Cut^1_w, Cut^1 or Str^1_u.

We argue that Π' can be rewritten into

$$\dfrac{\dfrac{\vdots \qquad \dfrac{\overset{\pi_1}{S^1 \to t} \quad \overset{\pi_2}{S, w\sigma \to p\sigma}}{S^1, S \to p\sigma}\, \mathsf{Cut}^2_w}{S^1, S \to p\sigma}\, \widetilde{R^1} \qquad \vdots}{S^n, S \to p\sigma}\, \widetilde{R^n}$$

This is a strictly smaller proof. It only remains to define the rules $\widetilde{R^i}$ and check that the above proof is a valid proof in the new inference system indeed.

$$\text{If} \qquad R^k = \dfrac{V_2^k \ \to_p t^k \quad V_1^k, w'\sigma \ \to_p t}{S^k \ \to_p t}$$

$$\text{we let} \qquad \widetilde{R_k} = \dfrac{V_2^k \ \to_p t^k \quad S, V_1^k, w'\sigma \to p\sigma}{S, S^k \to p\sigma}$$

The rule $\mathsf{Cut}^1_{w'}$ is therefore replaced with a rule $\mathsf{Cut}^2_{w'}$.

$$\text{If} \qquad R^k = \dfrac{V_1^k, v\sigma \to_p t}{V_1^k \to_p t} \qquad \text{we let} \qquad \widetilde{R^k} = \dfrac{S, V_1^k, v\sigma \to p\sigma}{S, V_1^k \to p\sigma}$$

The rule Str^1_v is replaced with a rule Str^2_v.

It is now enough to note that the choice of $\widetilde{R^k}$ ensures that Π' is a valid proof in the $\mathcal{I}$ inference system.

Computing Tiny Clause Normal Forms

Noran Azmy and Christoph Weidenbach

Max Planck Institute for Informatics

Abstract. Automated reasoning systems which build on resolution or superposition typically operate on formulas in clause normal form (CNF). It is well-known that standard CNF translation of a first-order formula may result in an exponential number of clauses. In order to prevent this effect, renaming techniques have been introduced that replace subformulas by atoms over fresh predicates and introduce definitions accordingly. This paper presents *generalized renaming*. Given a formula and a set of subformulas to be renamed, it is suggested to use one atom to replace all instances of a generalization of a given subformula. A generalized renaming algorithm and an implementation as part of the SPASS theorem prover are described. The new renaming algorithm is faster than the previous one implemented in SPASS. Experiments on the TPTP show that generalized renaming significantly reduces the number of clauses and the average time taken to solve the problems afterward.

1 Introduction

Most automated reasoning systems operate on formulas in *clause normal form* (CNF). Many problem formulations, however, are given in full propositional or first-order logic. While every first-order formula can be translated into an equisatisfiable formula in CNF [2], the conversion leads to an exponential blow-up of the formula in the worst case. An obvious example is this formula in *disjunctive normal form* (DNF),

$$(X_1 \wedge Y_1) \vee (X_2 \wedge Y_2) \vee \cdots \vee (X_n \wedge Y_n)\,,$$

with $O(n)$ literals, for which the CNF variant obtained by exhaustive application of distributivity laws has $O(2^n)$ literals.

A well-established workaround to this problem is *formula renaming*. Originally due to Tseitin [10], the idea is to introduce new predicate symbols to replace parts of the formula that might be "problematic", i.e. that might be eventually responsible for exponential blow-up of the result. These replacements avoid the repeated application of distributivity laws. For example, a renamed variant of the above formula is,

$$(R_1 \vee \cdots \vee R_n) \wedge (R_1 \leftrightarrow (X_1 \wedge Y_1)) \wedge \cdots \wedge (R_n \leftrightarrow (X_n \wedge Y_n))\,.$$

In this manner, the maximum number of new symbols introduced is in $O(n)$, and the CNF translation also becomes linear in the size of the formula. It is

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 109–125, 2013.

worth noting that renaming generates fewer clauses, without having an impact on the individual clause size. Because of the introduction of new symbols, formula renaming does not preserve logical equivalence. However, it is *satisfiability-preserving*; meaning that the resulting formula is satisfiable if and only if the original formula is satisfiable. This is not a problem in the context of automated theorem proving; CNF translation employs Skolemization and therefore is itself only satisfiability-preserving.

Existing work differs as to the criteria for choosing positions to rename. [8] suggests the replacement of every subformula, while [4] suggests only those replacements that would decrease the size of the CNF result. [7] shows an efficient method to find these "beneficial" replacements, which involves a linear-time computation of numbers and some arithmetic manipulation.

For first-order formulas, the authors of [7] mention the benefit of renaming several "compatible" subformulas using the same symbol. Two formulas φ_1, φ_2 are *compatible* if they are both instances of a common formula φ. We call such a φ a *generalization* of φ_1 and φ_2. For k compatible subformulas with a generalization φ, we need only define one symbol with the "general" definition φ, as opposed to k symbols that standard renaming would introduce. As an example, standard renaming of the formula,

$$(P \vee (Q_1(a_1) \wedge Q_2(a_1))) \wedge \cdots \wedge (P \vee (Q_1(a_n) \wedge Q_2(a_n))),$$

would yield the following formula of $4n$ clauses,

$$\begin{aligned} &(P \vee R_1) \wedge (P \vee R_2) \wedge \cdots \wedge (P \vee R_n) \\ &\wedge R_1 \leftrightarrow Q_1(a_1) \wedge Q_2(a_1) \\ &\wedge \ldots \\ &\wedge R_n \leftrightarrow Q_1(a_n) \wedge Q_2(a_n). \end{aligned}$$

Renaming of the compatible subformulas of the form $Q_1(a_i) \wedge Q_2(a_i)$ using one symbol yields a formula for which the CNF has only $n + 3$ clauses,

$$\begin{aligned} &(P \vee R(a_1)) \wedge (P \vee R(a_2)) \wedge \cdots \wedge (P \vee R(a_n)) \\ &\wedge \forall x\,[R(x) \leftrightarrow Q_1(x) \wedge Q_2(x)]. \end{aligned}$$

This inspires what we denote by *generalized formula renaming*. We assume the input to be a formula ψ, and a set Π of beneficial positions for renaming in ψ. For every beneficial position determined by the system, we would also like to rename all other compatible positions in ψ using the same predicate symbol.

Generalized renaming has several potential benefits. For redundant formulas, the clause set can be significantly reduced. New symbols are introduced minimally, and therefore there are fewer definition clauses. The individual clauses are also shortened by the replacements. Moreover, generalized renaming has the advantage of preserving structural information, which we will later show can have a significant effect on the time taken to find a proof.

This work is the first to study the generalized renaming problem. We present a formal generalized renaming algorithm with proof of correctness, and experiments based on our implementation of the algorithm as part of the SPASS automated theorem prover. Our tests show that generalized renaming is able to reduce the number of clauses in the CNF result by up to 92.9%, when compared to a standard renaming procedure where compatible subformulas are not considered. This significant reduction in size also reduces the time taken for CNF translation, and allows for potential reduction of the total run time of the prover. We will show cases where the performance of the prover is significantly enhanced with generalized renaming, and discuss the general effect of this approach on automated theorem proving.

We begin by fixing the notation and going over the necessary background in Section 2. Section 3 formally defines the problem and presents the algorithm. We show our experiments and results in Section 4, concluding with a discussion. All relevant proofs can be found in a technical report on [1].

2 Background

We rely mostly on the notation used in [7]. We assume a first-order language over a *signature* $\Sigma = (\mathcal{F}, \mathcal{R})$, where $\mathcal{F}$ and $\mathcal{R}$ are the sets of function and predicate symbols respectively. The *arity* of function (predicate) symbol f (P) is given by $arity(f)$ ($arity(P)$), with f/n denoting $arity(f) = n$. Additionally, we assume an infinite set $\mathcal{X}$ of *variable symbols* disjoint from the symbols in Σ. Terms, formulas, atoms, literals and clauses are defined in the usual way. A formula is in *clause normal form* if it is a conjunction of clauses. The set of positions of a term t (formula φ) is defined by: (1) $\epsilon \in pos(t)$ for any t, where $t|_\epsilon = t$, (2) if $t|_\pi = f(t_1, \ldots, t_n)$ for $f \in \mathcal{F}$, then $\pi.i$ is a position in t for all $i = 1, \ldots, n$, such that $t|_{\pi.i} = t_i$, (3) $\epsilon \in pos(\varphi)$ for any φ, where $\varphi|_\epsilon = \varphi$, (4) if $\varphi|_\pi = f(t_1, \ldots, t_n)$ for $f \in \mathcal{F}$, then $\pi.i$ is a position in φ for all $i = 1, \ldots, n$, such that $t\varphi|_{\pi.i} = t_i$, (5) if $\varphi|_\pi = P(t_1, \ldots, t_n)$ for $P \in \mathcal{R}$, then $\pi.i$ is a position in φ for all $i = 1, \ldots, n$, such that $\varphi|_{\pi.i} = t_i$, (6) if $\varphi|_\pi = \varphi_1 \odot \varphi_2$ for $\odot \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$, then $\pi.1$ and $\pi.2$ are positions in φ such that $\varphi|_{\pi.1} = \varphi_1, \varphi|_{\pi.2} = \varphi_2$, and (7) if $\varphi|_\pi = Qx\varphi'$ for $Q \in \{\forall, \exists\}$, or $\varphi|_\pi = \neg\varphi'$, then $\pi.1$ is a position in φ such that $\varphi|_{\pi.1} = \varphi'$.

We assume a *prefix order* $\leq$ on positions, with a corresponding strict ordering $<$. Two positions π_1, π_2 are *parallel*, denoted by $\pi_1 \parallel \pi_2$, if they are incomparable by $\leq$. In additional, there is the total lexicographic ordering $<_{lex}$. The size of a formula $|\varphi| = |pos(\varphi)|$. The *top symbol* of a term t (formula φ) is given by $topsymbol(t)$ ($topsymbol(\varphi)$). For a term t (formula φ), the function $vars(t)$ ($vars(\varphi)$) returns a set of variables that occur in t (φ). Similarly, $freevars(\varphi)$ is the set of free variables of φ.

The *polarity* of a position π in a formula ψ is denoted by $pol(\psi, \pi)$. Substitutions are defined in the usual way. We denote the identity substitution by σ_I. The *composition of two substitutions* σ, τ is denoted by $\sigma \circ \tau$. We denote the *most general unifier* by $mgu(t_1, t_2)$. For two formulas φ_1, φ_2, $subinst(\varphi_1, \varphi_2) := \{\pi \mid \varphi_2|_\pi = \varphi_1\sigma \text{ for some substitution } \sigma\}$. The application of a substitution σ

is extended to terms by the rule $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ for all $f \in \mathcal{F}$ with arity n. It is extended to formulas by (1) $P(t_1, \ldots, t_n)\sigma = P(t_1\sigma, \ldots, t_n\sigma)$ for every $P \in \mathcal{P}$, (2) $(\neg\varphi)\sigma = \neg(\varphi\sigma)$, (3) $(\varphi_1 \odot \varphi_2)\sigma = \varphi_1\sigma \odot \varphi_2\sigma$ for $\odot \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$, and (4) $(Qx\varphi)\sigma = Qy[(\varphi\{x \mapsto y\})\sigma]$ for $Q \in \{\forall, \exists\}$, where y is a fresh variable not in the domain or range of σ or $\mathit{freevars}(\varphi)$. An injective substitution σ for which $x\sigma \in \mathcal{X}$ for all $x \in \mathcal{X}$ is called a *variable renaming*. Syntactic equality on terms is defined in the usual way. For formulas, we define syntactic equality up to the names of quantified variables. Namely, for $Q \in \{\exists, \forall\}$, $Qx\varphi_1 = Qy\varphi_2$ if and only if $\varphi_1\{x \mapsto z\} = \varphi_2\{y \mapsto z\}$ for a fresh z. In all other cases, it is defined in the usual way.

We define formula renaming for first-order logic as in [7].

Definition 1. (Formula Renaming) *Let ψ, φ be first-order formulas, and π a non-term position such that $\psi|_\pi = \varphi$. Let $\mathit{freevars}(\varphi) = \{x_1, \ldots, x_n\}$, and R a fresh predicate symbol of arity n. A* formula renaming *of ψ at π is the first-order formula $\psi[\pi/R(x_1, \ldots, x_n)] \wedge \mathit{def}(\pi, \psi, R)$, where the formula $\mathit{def}(\pi, \psi, R)$ is a* polarity-dependent definition *of the new predicate R, defined as:*

$$\mathit{def}(\pi, \psi, R) := \begin{cases} \forall x_1, \ldots, x_n\,[R(x_1, \ldots, x_n) \rightarrow \psi|_\pi] & \text{if } \mathit{pol}(\psi, \pi) = 1, \\ \forall x_1, \ldots, x_n\,[\psi|_\pi \rightarrow R(x_1, \ldots, x_n)] & \text{if } \mathit{pol}(\psi, \pi) = -1, \\ \forall x_1, \ldots, x_n\,[R(x_1, \ldots, x_n) \leftrightarrow \psi|_\pi] & \text{if } \mathit{pol}(\psi, \pi) = 0. \end{cases} \quad (1)$$

The formula $\mathit{def}(\pi, \psi, R)$ defines the renaming predicate R in terms of $\psi|_\pi = \varphi$, according to the *polarity* of φ in ψ. This polarity-dependent definition is a well-known optimization, first suggested by [5], eliminating implications that are not useful for inferences. However, Equation 1 assumes that R is to be defined in terms of a subformula φ of ψ. In case of generalized renaming, we would like to define a renaming predicate R in terms of a *generalization* φ, where φ does not generally occur in ψ. We overload the function $\mathit{def}(\pi, \psi, R)$ with the following variant.

$$\mathit{def}(\varphi, p, R) = \begin{cases} \forall x_1, \ldots, x_n\,[R(x_1, \ldots, x_n) \rightarrow \varphi] & \text{if } p = 1, \\ \forall x_1, \ldots, x_n\,[\varphi \rightarrow R(x_1, \ldots, x_n)] & \text{if } p = -1, \\ \forall x_1, \ldots, x_n\,[R(x_1, \ldots, x_n) \leftrightarrow \varphi] & \text{if } p = 0, \end{cases} \quad (2)$$

where $\mathit{freevars}(\varphi) = \{x_1, \ldots, x_n\}$ and $p \in \{-1, 0, 1\}$. In this context, p refers to the polarity of the *instances* of φ. Because different instances of the same generalization may occur with different polarities in ψ, we use one "general" polarity which is compatible with all these instances. For a formula ψ, the polarity of $\Pi \subset \mathit{pos}(\psi)$ in ψ, $\mathit{pol}(\psi, \Pi) = p$ if $\mathit{pol}(\psi, \pi) = p$ for all $\pi \in \Pi$, $\mathit{pol}(\psi, \Pi) = 0$ otherwise.

For two formulas φ_1 and φ_2, we define the notion of generalization and compatibility as follows. A *generalization* of φ_1 and φ_2 is a formula φ such that for some substitutions σ_1, σ_2, $\varphi\sigma_1 = \varphi_1$ and $\varphi\sigma_2 = \varphi_2$. The set of all generalizations of φ_1 and φ_2 is denoted by $\mathrm{GEN}(\varphi_1, \varphi_2)$. Two formulas φ_1 and φ_2 are *compatible*, denoted by $\varphi_1 \sim \varphi_2$, if $\mathrm{GEN}(\varphi_1, \varphi_2) \neq \emptyset$.

In order to minimize the introduction of new variables, and consequently the arity of renaming predicates, we are interested in *minimal* generalizations. A generalization $\varphi \in \mathrm{GEN}(\varphi_1, \varphi_2)$ is called *minimal* if for all generalizations $\varphi' \in \mathrm{GEN}(\varphi_1, \varphi_2)$, there exists some σ such that $\varphi = \varphi'\sigma$. It is clear to see that minimal generalizations are equal up to variable renaming.

We extend the notion of generalization to sets of positions. Let ψ be a formula and $\Pi \subset pos(\psi)$. A *generalization of* Π *in* ψ is a formula φ such that for all $\pi \in \Pi$, there is some σ such that $\psi|_\pi = \varphi\sigma$. The set of all generalizations of Π in ψ is denoted by $\mathrm{GEN}(\psi, \Pi)$. A generalization $\varphi \in \mathrm{GEN}(\psi, \Pi)$ is called *minimal* if for every $\varphi' \in \mathrm{GEN}(\psi, \Pi)$ there is some σ such that $\varphi = \varphi'\sigma$.

Clearly, compatibility is an equivalence relation. For generalized renaming of compatible subformulas, we will cluster subformulas into equivalence classes with respect to compatibility. We denote these by *compatibility classes*. For a formula ψ, the *compatibility relation* $\sim_\psi$ on $pos(\psi)$ is an equivalence relation given by $\pi_1 \sim_\psi \pi_2$ if and only if $\psi|_{\pi_1} \sim \psi|_{\pi_2}$. The equivalence class of $\pi \in pos(\psi)$ with respect to $\sim_\psi$ is denoted by $[\pi]_\psi$. The *quotient set* of $pos(\psi)$ by $\sim_\psi$ is given by $Q_\psi := \left\{ [\pi]_\psi \mid \pi \in pos(\psi) \right\}$.

3 Generalized Formula Renaming

The input to the renaming algorithm is a formula ψ and a set of positions $\Pi \subset pos(\psi)$ to be renamed in ψ. Consider the formula,

$$\begin{aligned}\psi := \forall x, y\,[P(x) \leftrightarrow (Q(x) \leftrightarrow (P(y) \leftrightarrow (Q(y) \leftrightarrow (P(b) \leftrightarrow Q(b)))))] \\ \wedge\ ((P(a_1) \leftrightarrow Q(a_1)) \wedge \cdots \wedge (P(a_n) \leftrightarrow Q(a_n)))\end{aligned}$$

We assume that $\Pi = \{111122, 1111222, 11112222\}$, since renaming at these positions eventually reduces the number of clauses in the CNF transformation. A *standard renaming* of ψ at Π is the formula,

$$[\psi\,[\pi_i / R_i(x_1, \ldots, x_n)] \wedge def(\pi_i, \psi, R_i)]_{\pi_i \in \Pi},$$

where $\{x_1, \ldots, x_n\}$ are the free variables of $\psi|_{\pi_i}$. Applying standard renaming in our example yields the following formula,

$$\begin{aligned}&\forall x, y\,[P(x) \leftrightarrow (Q(x) \leftrightarrow R_2(y))] \\ \wedge\ &\forall x\,[R_2(x) \leftrightarrow (P(x) \leftrightarrow R_1(x))] \\ \wedge\ &\forall x\,[R_1(x) \leftrightarrow (Q(x) \leftrightarrow R_0)] \\ \wedge\ &[R_0 \leftrightarrow (P(b) \leftrightarrow Q(b))] \\ \wedge\ &(P(a_1) \leftrightarrow Q(a_1)) \wedge \cdots \wedge (P(a_n) \leftrightarrow Q(a_n))\end{aligned}$$

for which the CNF translation has $2n + 16$ clauses. Individual renaming of any of the subformulas $P(a_i) \leftrightarrow Q(a_i)$ does not reduce the number of clauses due to the need to introduce clauses for the definitions. However, we notice that the atom R_0 is used to rename a compatible subformula $P(b) \leftrightarrow Q(b)$, and can be used to rename all subformulas $P(a_i) \leftrightarrow Q(a_i)$ as well, without the introduction

of more definitions. Now, $P(b) \leftrightarrow Q(b), P(a_1) \leftrightarrow Q(a_1), \ldots, P(a_n) \leftrightarrow Q(a_n)$ are instances of $P(x) \leftrightarrow Q(x)$. We use this generalization to define the predicate R_0, which is now assigned an arity of 1. This *generalized renaming* yields,

$$\begin{aligned}
&\forall x, y\,[P(x) \leftrightarrow (Q(x) \leftrightarrow R_2(y))]\\
\wedge\ &\forall x\,[R_2(x) \leftrightarrow (P(x) \leftrightarrow R_1(x))]\\
\wedge\ &\forall x\,[R_1(x) \leftrightarrow (Q(x) \leftrightarrow R_0(b))]\\
\wedge\ &\forall x\,[R_0(x) \leftrightarrow (P(x) \leftrightarrow Q(x))]\\
\wedge\ &R_0(a_1) \wedge \cdots \wedge R_0(a_n)
\end{aligned}$$

for which the CNF translation has $n + 16$ clauses.

Formally, we define generalized renaming as follows.

Definition 2. (Generalized Formula Renaming) *Let ψ be a formula and $\Pi \subset pos(\psi)$ be a set of non-term positions in ψ. Then $\langle \psi, \Pi \rangle$ is a* formula renaming pair*. Let $sort_{lex}(\Pi) = [\pi_1, \ldots, \pi_n]$ be a list of the elements in Π, sorted in descending lexicographic order. A* generalized renaming *of Π in ψ is given by $rename(\psi, sort_{lex}(\Pi))$, where,*

$$\begin{aligned}
rename(\psi, \mathsf{NIL}) &= \psi\\
rename(\psi, [\pi_i \mid \Pi]) &= rename(rename(\psi, \pi_i), \Pi)\\
rename(\psi, \pi_i) &= \psi\left[\pi_i^j / R_i(x_1, \ldots, x_m)\,\sigma_i^j\right]_{\pi_i^j \in [\pi_i]_\psi} \wedge def(\varphi_i, p_i, R_i)
\end{aligned}$$

where $\varphi_i \in \mathrm{GEN}\left(\psi, [\pi_i]_\psi\right)$, $\psi|_{\pi_i^j} = \varphi_i \sigma_i^j$ $freevars(\varphi_i) = \{x_1, \ldots, x_m\}$, $p_i = pol\left(\psi, [\pi_i]_\psi\right)$ and R_i is a fresh predicate symbol of arity m.

Let $\psi_{\mathsf{SR}}, \psi_{\mathsf{GR}}$ be the CNF translations of ψ produced by standard and generalized renaming respectively. It is worth mentioning that whenever there is no redundancy to be exploited in the input formula ψ, i.e. whenever $[\pi]_\psi = \{\pi\}$ for all $\pi \in \Pi$, generalized renaming reduces to standard renaming of ψ at all positions $\pi \in \Pi$, and $\psi_{\mathsf{SR}} = \psi_{\mathsf{GR}}$.

In general, ψ_{GR} contains at most as many clauses as ψ_{SR}. It is possible, however, that generalized renaming results in more clauses than standard renaming by the above definition. When compatible instances of different polarities are included in the renaming, a definition may expand into a full equivalence where only an implication would suffice in the standard version. For example, if only position $\pi = 1$ is beneficial in $\psi := \phi\sigma_1 \wedge \neg\phi\sigma_2$, standard renaming would yield $A \wedge \neg\phi\sigma_2 \wedge \forall x_1, \ldots, x_n\,[A \rightarrow \phi\sigma_1]$ for some atom A, where $freevars(\phi\sigma_1) = \{x_1, \ldots, x_n\}$. Generalized renaming includes the compatible subformula $\phi\sigma_2$, yielding $B\sigma_1 \wedge \neg B\sigma_2 \wedge \forall y_1, \ldots, y_m\,[B \leftrightarrow \phi]$ for some atom B, where $freevars(\phi) = \{y_1, \ldots, y_m\}$, doubling the size of the definition. If we are interested in minimizing the number of clauses, generalized renaming may be

restricted to include only those compatible instances that will not expand the definition. Both types of generalized renaming are included in our SPASS implementation.

3.1 A Generalized Renaming Algorithm

Detecting compatible subformulas and constructing the appropriate definitions involves a process of repeated generalization. There are several issues to note here. First, every replacement of the form $\psi[\pi/R(x_1,\ldots,x_m)\sigma]$ requires the computation of the substitution σ. There are $|\psi|$ such substitutions in the worst case, each of size $O(|\psi|)$, which means that computing them requires quadratic time. To avoid this, we note that an upper replacement in the formula hides the effect of all lower replacements. Therefore, we need only perform those replacements at *top-most renaming positions*. A *top-most position in* ψ with respect to $\Pi \subseteq pos(\psi)$ is a position π such that for all $\pi' \in \Pi$, $\pi' > \pi$ or $\pi' \parallel \pi$. Top-most renaming positions are the top-most positions in Π. The sum of sizes of formulas at top-most renaming positions is at most linear.

However, nested renamings cause one renaming predicate to appear in the definition of another, which can be seen in our example, where $R_0(b)$ occurs in the definition of $R_1(x)$. This requires renaming steps to be performed in a bottom-up manner, as indicated in the definition. Compatibility checks on subformulas of a certain size can only be performed once all deeper subformulas (of smaller size) have been renamed. In order to employ top-most replacement, the definitions have to be constructed in such a way that is independent from the replacement steps. We will later show that, since dependencies between definitions correspond to similar dependencies between subformulas, we can use formula structure to compute the definitions correctly in a bottom-up manner, independent from the replacements.

We now present an algorithm for generalized renaming on a formula renaming pair $\langle \psi, \Pi \rangle$. Our algorithm performs replacements and generation of definitions in two separate stages.

The first step is to cluster subformulas of ψ into compatibility classes. The procedure described in COMPUTE-COMPATIBILITY performs this task. The result is a mapping M from subformulas to their respective generalizations.

COMPUTE-COMPATIBILITY relies on a process of repeated generalization. For this purpose, GENERALIZE describes a minimal generalization procedure for two input formulas φ_1, φ_2. Temporary substitutions are kept in σ_1, σ_2. A list L is used to keep track of bound variables, to prevent generalization above a bound variable position. It can be shown that the generalization returned by GENERALIZE is minimal.

At this point, every position $\pi \in pos(\psi)$ is assigned a minimal generalization $\varphi \in \mathrm{GEN}\left(\psi, [\pi]_\psi\right)$. Every such φ is assigned the polarity $pol\left(\psi, [\pi]_\psi\right)$, and φ is marked for renaming if and only if $[\pi]_\psi \cap \Pi \neq \emptyset$.

Let $\varphi_1, \ldots, \varphi_n$ be the set of constructed generalizations, where for $1 \leq i \leq n$, $\varphi_i \in \text{GEN}\left([\pi_i]_\psi\right)$ for some π_i, and φ_i is assigned polarity p_i and renaming predicate R_i. Because of possible nested renamings, R_i cannot be defined directly in terms of φ_i. In order to define R_i correctly, we need to detect what we call *direct dependencies* of φ_i on φ_j, where $j \neq i$, $1 \leq j \leq n$, and φ_j is marked for renaming. A *direct dependency* is the case where $\varphi_i|_\pi = \varphi_j\sigma$ for some π, σ such that $\varphi_i|_{\pi'} = \varphi_k\sigma'$ for no φ_k marked for renaming, $1 \leq k \leq n$, $k \neq i \neq j$ and $\pi' < \pi$. For every direct dependency $\varphi_i|_\pi = \varphi_j\sigma$, φ_i should be replaced at π with $R_j(x_1, \ldots, x_m)\sigma$, where $\mathit{freevars}(\varphi_j) = \{x_1, \ldots, x_m\}$.

Procedure: COMPUTE-COMPATIBILITY $(\langle \psi, \Pi \rangle, \pi)$

Input: A formula renaming pair $\langle \psi, \Pi \rangle$ and a position $\pi \in \mathit{pos}(\psi)$.
Result: A mapping M from subformulas of $\psi|_\pi$ to the respective generalizations.

if $\psi|_\pi$ *is a formula* **then**
  if $\psi|_\pi$ *is not an atom* **then**
    Recursively compute compatibility for subformulas of $\psi|_\pi$.
  end
  for *all generalizations* φ *computed so far* **do**
    Set $\langle \varphi, \sigma_1, \sigma_2 \rangle$ to GENERALIZE $(\psi|_\pi, \varphi, \emptyset, \emptyset, \emptyset)$.
    if $\varphi \neq \bot$ **then**
      Set $M(\psi|_\pi)$ to φ.
      if *the polarity assigned to* φ *is different from* $\mathit{pol}(\psi, \pi)$ **then** assign φ a polarity 0.
      if $(\pi \in \Pi)$ **then** mark φ for renaming.
      break.
    end
  end
  if *no generalization has been found for* $\psi|_\pi$ **then**
    Construct a new generalization $\varphi = \psi|_\pi$.
    Assign φ the polarity $\mathit{pol}(\psi, \pi)$.
    if $\pi \in \Pi$ **then** mark φ for renaming.
    Set $M(\psi|_\pi)$ to φ.
  end
end

Procedure: GENERALIZE $(\varphi_1, \varphi_2, \sigma_1, \sigma_2, L)$

Input: Formulas or terms φ_1, φ_2, substitutions σ_1, σ_2, and a list L of variables.
Output: $\langle \varphi, \sigma_1, \sigma_2 \rangle$ where $\varphi_i = \varphi\sigma_i$ for $i \in \{1, 2\}$ if $\varphi_1 \sim \varphi_2$, $\langle \bot, \mathsf{NIL}, \mathsf{NIL} \rangle$ otherwise.

if *topsymbol* (φ_1) = *topsymbol* (φ_2) **then**
  switch *topsymbol* (φ_1) **do**
    case *predicate/function symbol* P *where* $\varphi_1 = P(t_1, \ldots, t_n)$ *and* $\varphi_2 = P(s_1, \ldots, s_n)$
      Create list $A[1, \ldots, n]$.
      for $i = 1$ *to* n **do**
        Set $\langle t, \sigma_1, \sigma_2 \rangle$ to GENERALIZE $(t_i, s_i, \sigma_1, \sigma_2, L)$.
        if $t = \bot$ **then return** $\langle \bot, \mathsf{NIL}, \mathsf{NIL} \rangle$.
        $A[i] \leftarrow t$.
      end
      return $\langle P(A[1], \ldots, A[n]), \sigma_1, \sigma_2 \rangle$.
    case *negation symbol* $\neg$
      Set $\langle \varphi, \sigma_1, \sigma_2 \rangle$ to GENERALIZE $(\varphi_1|_1, \varphi_2|_1, \sigma_1, \sigma_2, L)$.
      if $\varphi = \bot$ **then return** $\langle \bot, \mathsf{NIL}, \mathsf{NIL} \rangle$.
      return $\langle \neg\varphi, \sigma_1, \sigma_2 \rangle$.
    case $\odot \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$
      Set $\langle \varphi'_1, \sigma_1, \sigma_2 \rangle$ to GENERALIZE $(\varphi_1|_1, \varphi_2|_1, \sigma_1, \sigma_2, L)$.
      Set $\langle \varphi'_2, \sigma_1, \sigma_2 \rangle$ to GENERALIZE $(\varphi_1|_2, \varphi_2|_2, \sigma_1, \sigma_2, L)$.
      if φ'_1 *or* φ'_2 *is* $\bot$ **then return** $\langle \bot, \mathsf{NIL}, \mathsf{NIL} \rangle$.
      return $\langle \varphi'_1 \odot \varphi'_2, \sigma_1, \sigma_2 \rangle$.
    case $Q \in \{\forall, \exists\}$ *where* $\varphi_i = Qx_i\varphi'_i$
      Rename x_i in φ'_i to $x \notin vars(\varphi_i) \cup dom(\sigma_i) \cup range(\sigma_i)$. Add x to L.
      Set $\langle \varphi, \sigma_1, \sigma_2 \rangle$ to GENERALIZE $(\varphi'_1, \varphi'_2, \sigma_1, \sigma_2, L)$.
      if $\varphi = \bot$ **then return** $\langle \bot, \mathsf{NIL}, \mathsf{NIL} \rangle$.
      return $\langle Qx\varphi, \sigma_1, \sigma_2 \rangle$.
    case *variable* x **return** $\langle x, \sigma_I, \sigma_I \rangle$.
  endsw
else if φ_1, φ_2 *are terms such that no* $x \in L$ *occurs in* φ_i **then**
  if *there is a variable* x *such that* $x\sigma_i = \varphi_i$ **then return** $\langle x, \sigma_1, \sigma_2 \rangle$.
  else
    Let x be a fresh variable not in $\varphi_i, dom(\sigma_i), range(\sigma_i)$.
    $\sigma_i \leftarrow \sigma_i \cup \{x \mapsto \varphi_i\}$.
    return $\langle x, \sigma_1, \sigma_2 \rangle$.
  end
else return $\langle \bot, \mathsf{NIL}, \mathsf{NIL} \rangle$.`// formulas with different top symbols`

Instead of searching for these dependencies in a naive way, we notice that for generalizations φ_1, φ_2 of $\psi|_{\pi_1}, \psi|_{\pi_1\pi_2}$ respectively, $\varphi_1|_{\pi_2} = \varphi_2\sigma$ for some σ. This is due to the property of *coverage*. A formula φ *covers* another formula ψ if for every formula φ' and substitution σ such that $\varphi = \varphi'\sigma$, $|\mathit{subinst}\,(\psi, \varphi)| = |\mathit{subinst}\,(\psi, \varphi')|$. Coverage can be shown to hold for generalizations constructed by our procedure (see technical report). Therefore, we may solve the dependencies by constructing definitions in a bottom-up manner, guided by the structure of ψ. At every position π in a bottom-up traversal of ψ, let φ be the generalization assigned to $\psi|_\pi$. We compute the definition def_φ for φ, as well as a *representative* rep_φ. If φ is marked for renaming, rep_φ is simply the renaming atom for φ. Otherwise, it is identical to def_φ. The definition def_φ can be easily computed using $\mathit{rep}_{\varphi'}$, where φ' are the generalizations of the immediate subformulas of $\psi|_\pi$.

Using our previous example, let $\varphi_1 = Q\,(x)\,, \varphi_2 = P\,(x) \leftrightarrow Q\,(x)$, and $\varphi_3 = Q\,(x) \leftrightarrow (P\,(y) \leftrightarrow Q\,(y))$ be generalizations constructed for $n+3$ and $n+1$ and 1 instance(s) in ψ, respectively. Because φ_1 is an atom unmarked for renaming, $\mathit{def}_{\varphi_1} = \mathit{rep}_{\varphi_1} = \varphi_1$. Now, φ_2 is marked for renaming with renaming predicate symbol R_0, and so $\mathit{def}_{\varphi_2} = P\,(x) \leftrightarrow Q\,(x)$, and $\mathit{rep}_{\varphi_2} = R_0\,(x)$. Similarly, φ_3 is marked for renaming with predicate symbol R_1, and $\mathit{rep}_{\varphi_3} = R_1(x)$. We construct $\mathit{def}_{\varphi_3} = \mathit{rep}_{\varphi_1} \leftrightarrow \big(\mathit{rep}_{\varphi_2}\,\{x \mapsto b\}\big)$, because $\varphi_3 = \varphi_1 \leftrightarrow \varphi_2\,\{x \mapsto b\}$.

Finally, Replace-Topmost performs top-most replacement, by traversing ψ in a top-down manner. If the current position π is a renaming position, it is replaced by the corresponding renaming atom. Otherwise, recursive calls are made to the immediate subformulas $\pi.1$ and $\pi.2$, if any.

4 Experiments

The generalized formula renaming algorithm is implemented as part of SPASS version 3.8 [11]. The TPTP problem library [9], version 5.4.0, was used for testing. Tests were run on a cluster of computing nodes with quad-core processors 2x Intel Xeon E5620 @ 2.40GHz, and 48GB RAM. The input set of beneficial renaming positions Π is computed using the same method in SPASS 3.7, suggested in [7].

The first experiment compared generalized renaming to standard renaming on all 7807 problems given in full first-order logic, from 36 different domains. In this experiment, only the CNF translation was performed, and the number of clauses and time taken for CNF translation was compared. For the second experiment, we considered the subset of these problems for which CNF translation finished within the given time limit, and generalized renaming was able to reduce the number of clauses by at least 5%. These are 528 problems from 16 different domains.

The CNF translation results can be found on the SPASS homepage [1].

Procedure: GENERATE-DEFINITIONS (ψ, π)

Input: A formula ψ and position $\pi \in pos(\psi)$.
Output: A conjunction δ of definition formulas for all generalizations of subformulas at or below $\psi|_\pi$.

Set δ to $\top$.
Let φ be the generalization assigned to $\psi|_\pi$.
if φ *is not yet defined* **then**
  switch *topsymbol* $(\psi|_\pi)$ **do**
    case *negation symbol* $\neg$
      Set δ to GENERATE-DEFINITIONS $(\psi, \pi.1)$.
      Set φ_1 to the generalization of $(\psi|_{\pi.1})$.
      Set def_φ to $\neg\left(rep_{\varphi_1}\right)\sigma$, where $\varphi = \neg\varphi_1\sigma$.
    case *one of* $\{\forall, \exists\}$
      Set δ to GENERATE-DEFINITIONS $(\psi, \pi.1)$.
      Set φ_1 to the generalization of $(\psi|_{\pi.1})$.
      Set def_φ to $Qx\left(rep_{\varphi_1}\right)\sigma$, where $\varphi = Qx\left[\varphi_1\sigma\right]$.
    case $\odot \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$
      Set δ to
      GENERATE-DEFINITIONS $(\psi, \pi.1) \wedge$ GENERATE-DEFINITIONS $(\psi, \pi.2)$.
      Set φ_i to the generalization of $(\psi|_{\pi.i})$ for $i \in \{1, 2\}$.
      Set def_φ to $\left(rep_{\varphi_1}\right)\sigma_1 \odot \left(rep_{\varphi_2}\right)\sigma_2$ for $\varphi = \varphi_1\sigma_1 \odot \varphi_2\sigma_2$.
    otherwise Set def_φ to φ. `// predicate symbol`

  endsw
  if φ *is marked for renaming* **then**
    Set rep_φ to $R(x_1, \ldots, x_n)$, where $freevars(\varphi) = \{x_1, \ldots, x_n\}$ and R/n is a new predicate symbol.
    Set δ to $\delta \wedge def\left(def_\varphi, p, R\right)$, where p is the polarity assigned to φ.
  else Set rep_φ to def_φ.
  Mark φ as defined.
end
return δ.

Procedure: REPLACE-TOPMOST (ψ, π)

Input: A formula ψ and position π.
Result: Every renaming position in $\psi|_\pi$ is replaced by the corresponding renaming predicate.

if $\psi|_\pi$ *is a non-atomic formula* **then**
  Let φ be the generalization assigned to $\psi|_\pi$.
  if φ *is marked for renaming* **then** set ψ to $\psi\left[\pi/\left(rep_\varphi\right)\sigma\right]$ for $\psi|_\pi = \varphi\sigma$.
  else recursively perform top-most replacement on subformulas of $\psi|_\pi$.
end

4.1 Comparing the Size of CNF

In this experiment, we performed CNF translation on 7807 TPTP problems using both standard renaming (SR), and generalized renaming (GR). The respective number of clauses and time taken for CNF translation were compared. The number of problems for which both versions terminated within the time limit of one hour is 7231, leaving 576 particularly large problems. This is partly due to the fact that the SPASS 3.8 implementation is not targeted towards very large input files in general.

Fewer Clauses: Generalized renaming reduces the number of clauses in the CNF result for 1983 (27.4%) of the problems. Clause reduction may be as high as 93%, as in problem ALG207+1. In this problem, 147 occurrences of subformulas of the form $(\mathsf{op}\,(x_1, y_1) \approx z_1) \vee \cdots \vee \mathsf{op}\,(x_7, y_7) \approx z_7$ are replaced by instances of $\mathsf{SkP0}\,(x_1, y_1, z_1, \ldots, x_7, y_7, z_7)$, reducing the number of clauses from 1098 to only 78. Average clause reduction over each problem domain also proves significant, reaching 15.4% for some domains (Table 1a). For 528 problems, there is at least 5% reduction in the number of clauses. We focus on these problems in our theorem proving experiment in the next section. Table 1b shows clause reduction for hard problems. For some of the hardest TPTP problems (with rating 1.0), generalized renaming was able to reduce the number of clauses by up to 87%. Although this reduction in size does not yet enable SPASS to solve these problems, the significant reduction in size means that generalized renaming is able to discover a hidden structure in these problems. Developing new reasoning mechanisms for the abbreviated structure is a new approach towards tackling these problems in the future (see our discussion on theorem proving mechanisms in the next section).

Less CNF Translation Time: On average, generalized renaming does not require more time for CNF translation. In fact, while the average improvement in CNF translation time is 0.6%, the time taken for CNF translation is significantly reduced in many cases. Naturally, this depends on the structure of the problem, the applicability of generalized renaming, compared to the overhead required for compatibility computation. Both the maximum improvement and deterioration in CNF translation time are around 30%, for problems CSR086+2 and CSR104+2 respectively.

4.2 Comparing Theorem Proving Time

For this experiment, SPASS was run on all 528 problems for which there was at least 5% clause reduction, to test if this significant size improvement corresponds to an improvement in proof times. For these problems, there was 30.9% average clause reduction, and 37.8% average reduction in CNF translation time. The time limit was set to 1 day. The total run time of the prover was compared.

For the problems solved by both versions, generalized renaming is able to outperform standard renaming, as shown in Table 1. The time taken for theorem proving is improved by 59.62%.

Table 1. Some of the best results according to problem domain/difficulty

(a) Problem domains with significant clause reduction.

Domain	Clauses Saved	CNF Time Saved
SWV	15.40%	93.14%
LCL	15.22%	16.28%
ALG	12.72%	3.87%
COM	3.35%	4.32%
MSC	2.93%	6%

(b) Hard problems with significant clause reduction.

Problem Count	Rating	Maximum Clauses Saved
32	1.0	86.9%
9	$[0.9, 1.0[$	21.7%
21	$[0.8, 0.9[$	23.8%
28	$[0.7, 0.8[$	85.5%
27	$[0.6, 0.7[$	85.7%
62	$[0.5, 0.6[$	67.6%

Table 2. Problems where SPASS terminates within the time limit of 1 day, using both renaming versions. Generalized renaming significantly outperforms standard renaming with respect to theorem proving.

Problem Count	Average CNF Size		Average Time		Time Improvement
	SR	GR	SR	GR	
381	287.88	194.90	294.39	118.88	59.62%

However, the effect of generalized renaming on the time taken to find a proof for a specific problem is rather complex. Generalized renaming has two main effects on the resulting clause set. First, generalization results in clauses with more variables. This has many consequences. For one, ground clauses produced by standard renaming may correspond to non-ground ones produced by generalized renaming, which means that certain mechanisms such as splitting or reduction may no longer be possible. An example of this is problem ALG091+1, where generalized renaming succeeds in reducing the number of clauses by 90.5%. However, the CNF result after generalized renaming contains the clause $(\mathsf{op}\,(u, u) \approx v \vee \mathsf{SkP0}\,(w, x, y, v, u))$. This clause cannot be split, because of the shared variables between the two literals. It corresponds, however, to several ground clauses in the standard renaming version, each of which can be split. In this case, splitting is essential to finding the proof. A mechanism of splitting together with instantiation is needed, as suggested in [6]. However, this has not yet been implemented.

It may also happen that some reductions performed using standard renaming are not performed in the generalized version. For example, for ALG047+1 a ground reduction turns into an instance of contextual rewriting after translation

with generalized renaming. As this instance of contextual rewriting is currently not implemented in SPASS, the problem can no longer be solved.

Second, the ordering of literals within a clause in one version may be different from that within the corresponding clause in the other version, due to the reduction in size of literals at renaming positions. This means that some inferences may no longer be possible, and vice versa, because the maximal literal in the clause is different. When a renaming predicate SkP is introduced with an equivalence formula as a definition $\mathsf{SkP} \leftrightarrow \varphi$, there are essentially two proof directions for the prover to take. One direction is to apply inferences involving SkP in order to *expand* the definition, i.e. revert the renaming. The other direction involves inferences that do not revert the renaming, but use information about SkP to find the proof. It is clear that these two proof attempts are complementary. While the first direction is suited for some problems, the second may be better-suited for a different class of problems. An example of this is problem NLP160+1, where the prover sidetracks to irrelevant inferences involving the renaming predicate $\mathsf{SkP2}$, marked as maximal in the respective clauses. By enforcing an ordering in which $\mathsf{SkP2}$ is no longer maximal, the problem is solved.

Third, the change in the individual clause size and depth means that clauses are selected in different order by the prover. It may happen that useful clauses are only considered much later (or vice versa). In short, slight changes in the shape of the input clause set may lead the prover down a very different path. The outcome depends mostly on the problem, but also on the approach/heuristics used by the prover.

Based on the above, we have implemented two additional settings in our generalized renaming technique. The first setting, denoted by GC, prevents generalized renaming of closed subformulas (i.e. subformulas with no free variables). For any set of pairwise-compatible subformulas with no free variables, each subformula is renamed separately, i.e. only if its renaming is beneficial with respect to the number of clauses. This setting is well-suited for a number of TPTP problems where ground reductions are particularly helpful in finding a proof.

The second setting, denoted by GMX, enforces proof attempts that do not expand renaming predicate definitions. When enabled, renaming predicates whose definitions are expanded from an implication into a full equivalence by generalized renaming are assigned a minimal ordering with respect to other predicates.

Testing each setting individually against standard renaming on all 528 problems, we get the results shown in Table 3. Although SR manages to solve the highest number of problems in total, it solves fewer problems than the three versions of generalized renaming combined. This is because the different flavors of generalized renaming are suited for different types of input problems. For example, the 391 problems solved by plain GR are not the same as those solved by GC. These initial results suggest that a more robust renaming procedure should combine all these different versions.

By running a simple portfolio of GC for 30 seconds, GMX for 10 seconds and plain generalized renaming for the rest of the timeout of 1 day, we obtain the results summarized in Tables 4 and 5. Out of the problems solved by standard

Table 3. Performance of standard renaming along with three versions of generalized renaming.

	Solved	Unsolved
SR	414	114
Plain GR	391	137
GC	391	137
GMX	402	126

Table 4. Problems where SPASS terminates within the time limit of 1 day, using both renaming versions with GC and GMX settings. The generalized renaming ensemble significantly outperforms simple standard renaming.

Problem Count	Average Time		Time Improvement
	SR	GR/GC/GMX	
409	281.77	145.42	48.4%

Table 5. Problems where SPASS terminates within the time limit of 1 day, using both renaming versions with GC and GMX settings. SPASS does not terminate using simple SR.

Problem	Time (GR/GC/GMX)	Problem	Time (GR/GC/GMX)
LCL649+1.010	22.51	LCL659+1.020	0.3
LCL659+1.005	0.07	LCL660+1.005	1439.17
LCL659+1.010	0.17	LCL660+1.010	15080.06
LCL659+1.015	0.24	SWV024+1	2359.38

renaming, generalized renaming only fails to solve one problem, LCL684+1.005. This problem is solved by the standard procedure in 30.14 seconds. On the other hand, our technique is able to solve 8 problems that cannot be solved using standard renaming. Out of 528 problems, 110 remain unsolved by both procedures.

5 Conclusion

This paper provides a formal definition, algorithm and implementation for the generalized renaming problem. Given a formula ψ, our goal is to obtain a shorter

CNF result, by renaming ψ to an equisatisfiable variant ψ'. Instead of renaming every subformula of ψ, we assume a set Π of *renaming positions* is given as input. We rename ψ at all the given renaming positions, as well as positions of *compatible subformulas*, which can be renamed at no additional cost. We define compatible formulas as formulas that are instances of a common formula. Of course, our generalized renaming technique is also applicable to the translation of propositional formulas and SAT solving [3]. This is also an interesting starting point for further experiments, because in SAT solving the strengthening of the polarity of a definition often speeds up a subsequent solver run.

Our experiments show that generalized renaming reduces the number of CNF clauses by up to 92.9%. For some problem domains, the average clause reduction is as high as 15.4%. The time taken for CNF transformation is also significantly reduced in the average case. More importantly, generalized renaming works for some cases for which the standard renaming procedure fails. Although the time taken to find a proof is improved by up to 59.62% in the general case, the effect of introducing more variables is rather complex and may lead to an increase in time. With the introduction of some new mechanisms, however, our technique is able to outperform the standard procedure, with a time improvement of 48.4%.

Limitations and Future Work. Technically, logical operators may have arbitrary number of arguments. Suppose some subformula ϕ is to be renamed in $\psi := \phi_1 \wedge \cdots \wedge \phi_i \wedge \phi_{i+1} \wedge \cdots \wedge \phi_n$, represented as one conjunction operator with n arguments. If $\phi_i \wedge \phi_{i+1} \sim \phi$, this is not detected by our algorithm; the conjunction $\phi_i \wedge \phi_{i+1}$ does not exist from a technical point of view. In order to handle such cases, compatibility computation should rely on a more sophisticated pattern-matching procedure on term trees, instead of subformula checks. The algorithm also not aware of associativity/commutativity. If some subformula ϕ of $\psi := \phi_1 \wedge (\phi_2 \wedge \phi_3)$ is to be renamed and $\phi \sim \phi_1 \wedge \phi_2$, this compatible instance is not found in ψ. Since this is an NP problem, an efficient implementation modulo associativity/commutativity must resort to heuristics or approximations that provide accurate results without compromising efficiency.

As discussed in Section 4.2, there is potential to find more proofs/model by adapting the inference and reduction mechanisms in SPASS to the structure of the clauses generated by generalized renaming. For example, new variants of contextual rewriting can be implemented and dedicated to the renaming predicates introduced by generalized renaming.

References

1. SPASS Current Prototypes and Experiments, `http://www.spass-prover.org/prototypes/index.html`
2. Baaz, M., Egly, U., Leitsch, A.: Normal Form Transformations. In: Voronkov, A., Robinson, A. (eds.) Handbook of Automated Reasoning, ch. 5, pp. 273–333
3. Biere, A., Heule, M., Maaren, H.V., Walsh, T. (eds.): Handbook of Satisfiability. IOS Press (2009)

4. Thierry Boy de la Tour. An Optimality Result for Clause Form Translation. Journal of Symbolic Computation 14(4), 283–301 (1992)
5. Henschen, L., Lusk, E., Overbeek, R., Smith, B.T., Veroff, R., Winker, S., Wos, L.: Challenge Problem 1. SIGART Newsletter (72), 30–31 (1980)
6. Hillenbrand, T., Weidenbach, C.: Superposition for bounded domains. In: Bonacina, M.P., Stickel, M.E. (eds.) McCune Festschrift. LNCS (LNAI), vol. 7788, pp. 68–100. Springer, Heidelberg (2013)
7. Nonnengart, A., Weidenbach, C.: Computing Small Clause Normal Forms. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 6, pp. 337–367 (2001)
8. Plaisted, D.A., Greenbaum, S.: A Structure-preserving Clause Form Translation. Journal of Symbolic Computation 2(3), 293–304 (1986)
9. Sutcliffe, G., Suttner, C.: The TPTP Problem Library for Automated Theorem Provers (September 2010), `http://www.tptp.org/`
10. Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus. Studies in Constructive Mathematics and Mathematical Logic 8(115-125), 234–259 (1968)
11. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS Version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 140–145. Springer, Heidelberg (2009)

System Description: E-KRHyper 1.4

Extensions for Unique Names and Description Logic

Markus Bender, Björn Pelzer, and Claudia Schon

Universität Koblenz-Landau, Institut für Informatik, 56070 Koblenz, Germany
{mbender,bpelzer,schon}@uni-koblenz.de

Abstract. Formal ontologies may go beyond first-order logic (FOL) in their expressivity, hindering the usage of common automated theorem provers (ATP) for ontology reasoning. The Unique Name Assumption (UNA) is an extension to FOL that is valuable for ontology specification, allowing the definition of distinct objects. Likewise, the Description Logic $\mathcal{SHIQ}$ is a popular language for knowledge representation (KR). This system description provides details on the extension of the prover E-KRHyper by the ability to handle both the UNA and $\mathcal{SHIQ}$. This ATP was developed for embedding in KR applications and hence already equipped with special features and extensions to FOL, making it natural to add the new capabilities in E-KRHyper version 1.4. We report on the theory, the implementation and also the evaluation results of the new features.

1 Introduction

The expressivity of first-order logic (FOL) is often insufficient when specifying ontologies. This has led to logic extensions like the *Unique Name Assumption* (UNA), which allows expressing that two constants are semantically distinct. Going further than merely extending FOL, Description Logics provide entire logical languages intended especially for knowledge representation (KR). Unfortunately automated theorem provers (ATP) are commonly restricted to FOL, which limits their usefulness for ontology reasoning. We have therefore modified our prover *E-KRHyper*, giving it the capability to handle both the UNA and the popular Description Logic $\mathcal{SHIQ}$.

E-KRHyper (*Knowledge Representation Hyper Tableaux with Equality*) [31] is an ATP and model generation system for FOL with equality. It is an implementation of the *E-hyper tableau calculus* [12], which integrates a superposition-based handling of equality [4] into the hyper tableau calculus [11]. Like its predecessor, the original KRHyper [39] without equality, E-KRHyper has been designed for embedding in knowledge representation applications, see Section 2 for details.

E-KRHyper forms the reasoning component of the deduction-based question answering system *LogAnswer* [17,18]. The prover is also used in the controlled language processor *PENG Light* [35], and it is part of the *HETS* framework [26] for formal methods integration and proof management. (E-)KRHyper has also been used as an embedded knowledge processing engine in applications like content composition for e-learning [6,9], document management [8], database schema processing [7], semantic information retrieval [5], ontology reasoning [14], and planning [13]. An excerpt has been ported to mobile devices for user profile matching [36].

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 126–134, 2013.

Given this background we wish to keep improving E-KRHyper's abilities in ontology reasoning, so it was natural to soundly integrate the UNA [15] and $\mathcal{SHIQ}$ [16] both on the formal level and in the implementation. In this system description we provide details on the current version 1.4 of E-KRHyper, focussing on the new extensions, and we evaluate the system on suitable benchmarks.

2 General Overview of E-KRHyper

E-KRHyper 1.4 is written in OCaml [24] and compatible with POSIX-compliant operating systems and environments. It is available under the GNU GPL at [29]. As E-KRHyper is geared towards embedding, it has been designed with operational flexibility: like most ATP systems it can be invoked with file-based input, but it also features an interactive mode in which it can communicate with applications or the user over STDIN and STDOUT or other user-specified channels. This way E-KRHyper can remain operational over multiple reasoning tasks, with the logical input being modified in-between, for example in order to test different conjectures on a common knowledge base without having to reload the latter each time [30,19].

The prover accepts input in *PROTEIN* [10] and *TPTP* syntax [38] for FOL. A built-in clausifier automatically converts formula expressions into CNF. To deal with very large input theories E-KRHyper includes axiom selection methods like the SINE algorithm [21]. Input may go beyond the expressivity of FOL with equality: E-KRHyper supports stratified negation as failure [1], arithmetic evaluation as well as operations on list and set constructs. Special interface literals allow the prover to access external knowledge sources like web services or databases.

The proof procedure involves the construction of an E-hyper tableau, which is a tree labelled with clauses. A branch may be extended by applying an inference rule to its clauses and then appending the conclusion as a new leaf node. A branch is closed once it contains the empty clause. Positive disjunctions can split branches. Ground substitutions prevent sharing of variables between disjunct branches. This avoids the issue of rigid variables, and it allows E-KRHyper to build the tableau depth-first, working only on one branch at a time. Refutational completeness and a fair search control are ensured by an iterative deepening strategy bounded by the term weights of clauses. A comprehensive set of simplification and deletion rules helps the prover to eliminate redundant clauses and to avoid unnecessary inferences. If all branches are closed, E-KRHyper can output a proof consisting only of the essential inference steps. On the other hand, a branch that can be neither closed nor extended represents a model and proves the input satisfiable. E-KRHyper can then enumerate models by computing additional branches. The derivation process can also be refined to allow the computation of minimal models.

3 Extensions in Version 1.4

3.1 Support for the Unique Name Assumption

In automated theorem proving there are problems that need information on the inequality of certain constants [2,20,3]. In most cases such information about any two distinct constants c_1 and c_2 is provided by explicitly adding inequality facts of the form

false $\leftarrow c_1 \simeq c_2$ to the knowledge base. As the number of these facts grows quadratically in the number of constants, they clutter the knowledge base and distract human readers of the problem from its actual proposition. Additionally it is safe to assume that a larger knowledge base reduces the performance of a theorem prover in many cases, which is another drawback of explicit inequality facts.

To avoid such a blow-up, native handling of inequality of constants can be used, which removes the need for inequality facts. This can be done by using the *Unique Name Assumption* (UNA), which states that two constants are semantically equal if and only if they are syntactically equal. Instead of forcing the UNA onto all constants, we apply it on a subset of the constants, called the *distinct object identifiers* (DOI). This is a more flexible approach and mimics the TPTP's way of dealing with the UNA [38,37]. We also use the TPTP's syntax for defining DOI, i.e. a constant enclosed in double quotes is treated as a DOI by E-KRHyper.

E-KRHyper's underlying calculus was extended in a sound and correct way to allow native handling of DOI and those changes have been implemented into E-KRHyper accordingly [15]. The most significant change was the introduction of two rules to deal with two kinds of formulæ that are important if DOI are used: *unit contradictions*, of the form $i \simeq j \leftarrow$, and *object tautology clauses*, of the form $\mathcal{A} \leftarrow i \simeq j, \mathcal{B}$, where i and j are two non-identical DOI, $\mathcal{A}$ [$\mathcal{B}$] is a possibly empty set of head [body] literals.

An unit contradiction is a clause that is false by definition, so the modified version of E-KRHyper closes a branch that contains a unit contradiction. On the opposite site, an object tautology clause is always true and thus can neither contribute to deriving a contradiction nor a model. Therefore object tautology clauses are deleted to prevent unneeded inference steps on the clause and to keep the set of clauses as small as possible.

This implicit handling of non-identical constants makes the problems easier to comprehend and reduces the execution time of reasoning.

3.2 Support for $\mathcal{SHIQ}$

$\mathcal{SHIQ}$ is a description logic (DL) with widespread usage. It is decidable and features both inverse and transitive roles as well as qualified cardinality restrictions. Thanks to this it has become the formal foundation of OIL (Ontology Inference Layer), the ontology infrastructure of the Semantic Web [23]. Several dedicated DL-reasoners support $\mathcal{SHIQ}$, such as *Pellet* [28] and *HermiT* [27]. The latter even employs a modified version of our own hyper tableau calculus. Adapting E-KRHyper to $\mathcal{SHIQ}$ was therefore an obvious way for us to enhance the ontology reasoning abilities of our prover.

E-KRHyper accepts $\mathcal{SHIQ}$ converted into *DL-clauses* [27], which are basically FOL-clauses extended by the possibility of having special positive *at-least* literals which represent qualified number restrictions. For example, we can express that a car has at least three wheels with the $\mathcal{SHIQ}$-axiom $car \sqsubseteq (\geq 3\ has.wheel)$, which translates into the DL-clause $(\geq 3\ has.wheel)(x) \leftarrow car(x)$. Hence an *at-least* literal has the form $(\geq n\ R.C)(x)$, with $n \geq 1$ and where R is a binary predicate symbol (representing the DL-role R), C is a unary (and possibly negated) predicate (representing the DL-concept C), and x is a variable (to be instantiated by ground terms that represent DL-individuals). E-KRHyper has been modified to handle such DL-clauses, and a corresponding *at-least* inference rule has been added to the calculus to evaluate the new literals:

$$\textit{at-least}\frac{(\geq n\ R.C)(a) \leftarrow}{\bigcup_{1\leq i\leq n}\{R(a,b_i) \leftarrow\} \cup \bigcup_{1\leq i\leq n}\{C(b_i) \leftarrow\} \cup \bigcup_{1\leq i<j\leq n}\{\leftarrow b_i \simeq b_j\}}$$

Thus given as a premise a positive *at-least* unit instantiated by a ground term a, the rule creates n fresh ground terms $b_1, \ldots, b_n$ (the *R-successors* of a) and derives unit clauses for these new individuals, ensuring that they satisfy the predicates R and C and that they are pairwise distinct. The derived set is considered a conjunction, so all its clauses are attached to the current branch as a segment of $2n + \binom{n}{2}$ nodes. To ensure termination it may be necessary to block an application of the *at-least* rule, to prevent it from generating useless successors for a given individual. For this we use a modified version [16] of *pairwise anywhere blocking* [27].

As an optional optimization E-KRHyper can use its UNA functionality (see Section 3.1): instead of ensuring the distinctness of successors via explicit inequalities, the fresh $b_1, \ldots, b_n$ created by one application of the *at-least* rule can be treated as a special type of DOI which only count as distinct with respect to each other, but which are treated as normal terms otherwise. For example, $b \simeq b' \leftarrow$ for some generated successor terms b and b' is only a unit contradiction if both b and b' have been created in the same *at-least* inference step.

4 Related Systems and Performance Evaluation

We have evaluated several aspects of E-KRHyper: the performance on FOL-problems in general, and specifically the performance of the two new extensions, namely the UNA implementation and the $\mathcal{SHIQ}$ capability. All tests were carried out on a computer featuring an Intel Core 2 Quad (Q9550) @ 2.83GHz and 4GB PC2-6400 RAM.

For the general evaluation we tested E-KRHyper on the 15,560 FOL problems of the TPTP v5.4.0, the most current release at the time of this writing. The prover was limited to one CPU core, 1GB of memory and 300 seconds for each problem. E-KRHyper solved 5,970 problems, corresponding to 38.4% of the test set.[1] The TPTP rates the difficulty of problems from 0.0 for the easiest to 1.0 for the hardest, the latter reserved for problems not solved by any state-of-the-art prover. The hardest problems solved by E-KRHyper were *PUZ050-1* (rated at 0.94), *SWW284+1* (0.93) and *LCL650+1.020* (0.92). Overall E-KRHyper has a general performance that is slightly above average among the provers listed at the TPTP website [37]. In comparison, the prover *Otter* [25], which usually serves as a benchmark in the ATP community, solves 27% under similar conditions, whereas top-ranking systems like *Vampire* [32] and *E* [33] have a significant lead with 66% and 61% respectively.

4.1 Unique Name Assumption

In order to evaluate the UNA implementation we used a variation of the synthetic benchmarks STORECOMM (SC) and STORECOMM-INVALID (SCI) [2], situated in the

[1] The TPTP v5.4.0 contains merely two FOL problems with DOIs and none with DL-features, so the prover modifications described in this paper lead only to a negligible improvement in the TPTP performance compared to earlier versions of E-KRHyper.

theory of arrays. A test case from SC is the task to show that two given permutations of unique store operations on an array result in the same array. A test case from SCI is the task to show that two given sequences of unique store operations that differ in at least one stored element at an index do not result in the same array. This definition of SCI differs from the definition in [2] and was chosen for technical reasons. We continue to call it STORECOMM-INVALID or SCI in the context of this work. The term *unique store operation* states that each index of an array is written to exactly one time.

As the theory of arrays is not natively supported by E-KRHyper, we axiomatize the theory of arrays by using the following axioms:

$$sel(sto(A, I, E), I) = E \tag{1}$$

$$sel(sto(A, I, E), J) = sel(A, J) \Leftarrow I \neq J \tag{2}$$

$$A = B \Leftarrow sel(A, I) = sel(B, I) \tag{3}$$

The function $sel : ARRAY \times INDEX \rightarrow ELEMENT$ returns the element that is stored at the given index of the given array, and the function $sto : ARRAY \times INDEX \times ELEMENT \rightarrow ARRAY$ returns an array that is constructed by storing a given element at the given index of a given array.

To create a test case, four parameters are needed: a list $p = 0, \ldots, n-1$, a permutation of this list, called q, a flag v that indicates if we want to generate a test case for SC or SCI and a flag d that indicates if this test case uses distinct object identifiers or not.

Independent of the chosen parameters every test case contains the three axioms that describe the theory of arrays. Additionally every test contains n unique predicates of the form $index(i_x)$ with $0 \leq x < n$ introducing the constants that represent the arrays' indices, which is needed for technical reasons. If distinct object identifiers are used, these predicates look like $index(\text{"}i_x\text{"})$, as constants in quotes denote DOI.

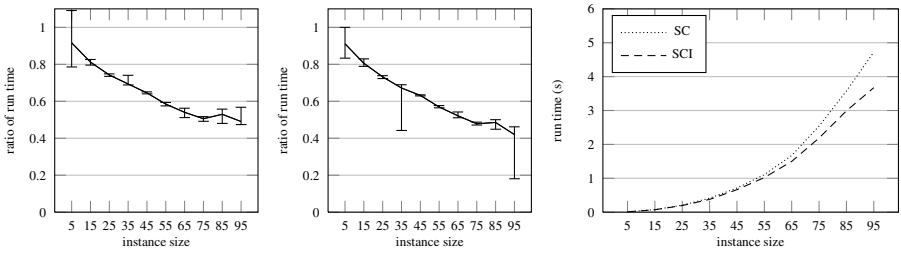
If no distinct object identifiers are used, we need to express that all indices are distinct, which is done by introducing $\binom{n}{2}$ unique predicates of the form *false* :- $i_x = i_y$ with $(x, y) \in C_2^n$, where C_2^n is the set of 2-combinations over $\{0, \ldots, n-1\}$. Additionally we need to express that all elements are distinct, which is done by introducing $\binom{n}{2}$ unique predicates of the form *false* :- $e_x = e_y$ with $(x, y) \in C_2^n$.

The actual property to be proven is then added by the equality predicate $T_{n,v,d}(q) = T_{n,v,d}(p)$, where $T_{k,v,d}(l)$ is defined as follows:

$$T_{k,v,d}(l) = \begin{cases} a & \text{if } k = 0 \\ sto(T_{k-1,v,d}(l), i_{l(k)}, e_0) & \text{if } k = 1 \text{ and } v = 0 \text{ and } d = 0 \\ sto(T_{k-1,v,d}(l), \text{"}i_{l(k)}\text{"}, \text{"}e_0\text{"}) & \text{if } k = 1 \text{ and } v = 0 \text{ and } d = 1 \\ sto(T_{k-1,v,d}(l), i_{l(k)}, e_{l(k)}) & \text{if } 0 \leq k < n \text{ and } v = 1 \text{ and } d = 0 \\ sto(T_{k-1,v,d}(l), \text{"}i_{l(k)}\text{"}, \text{"}e_{l(k)}\text{"}) & \text{if } 0 \leq k < n \text{ and } v = 1 \text{ and } d = 1 \end{cases}$$

We generated four different types of test cases: SC[I] without DOI, and SC[I] with DOI and considered arrays with $5, 15, \ldots, 95$ elements. For each of the 40 type-size-combinations, 20 random instances were generated and then flattened.

For an easier comparison of the results with and without DOI, they were put into relation by dividing the runtimes with DOI by the according runtimes without DOI. The



(a) Run time gain for SC (b) Run time gain for SCI (c) Run times for SCI and SC using DOI

Fig. 1. Relative execution times for S-COMM and S-COMM-INV

Table 1. Comparison of benchmark results for E-KRHyper, HermiT and Pellet

Ontology Name	Nr. of Axioms	Runtimes (s)				
		Transformation	E-KRHyper (w/o DOI)	E-KRHyper (with DOI)	HermiT	Pellet
galen	5134	7.44	2.761	2.755	4.685	25.075
galen-ians-full-doctored	5668	8.70	2.987	2.992	4.887	28.102
galen-ians-full-undoctored	5911	8.27	3.152	3.148	7.596	29.441
not-galen	6825	13.61	3.570	3.565	11.057	28.665
emap	13730	19.09	4.270	4.259	10.894	64.533
vicodi_0	53876	11.79	2.699	2.675	10.522	106.831
vicodi_1	107529	16.26	6.941	6.893	12.632	233.324
vicodi_2	161182	18.50	13.246	13.221	15.074	387.200
vicodi_3	214835	20.05	19.092	19.201	17.719	512.019
vicodi_4	268488	24.95	24.411	24.590	18.819	ERROR

result of this operation can be seen in Fig. 1a [Fig. 1b] for SC [SCI], where the graph shows the average, minimal and maximal runtime, for each size. It is easy to see that the version with DOI outperforms the one without DOI and that the difference grows with the size of the arrays. We assume that the specific structure of the problems influences the runtime, which might lead to the visible fluctuations in the graphs. This has not yet been investigated. We compared the results of the DOI version of E-KRHyper with the results of the prover E, which is also able to handle DOI [34]. E solved the 200 problems for SC [SCI] with an average runtime of 0.09 [0.08] seconds where E-KRHyper needed 1.5 [1.27] seconds. Fig. 1c shows that for E-KRHyper with DOI the runtime in relation to the array size is quadratical. [15] provides details on the setup, benchmarking process and the evaluation.

4.2 Description Logic

For an evaluation E-KRHyper's performance on DL-ontologies we have chosen several ontologies from [22] and checked them for consistency. Systems for comparison are HermiT and Pellet. Tab. 1 shows the runtimes for those ontologies and provers.

The column *Transformation* lists the times needed to transform the ontologies into DL-clauses, i.e. a suitable input for E-KRHyper. Even with adding the transformation and inference times, E-KRHyper outperforms Pellet significantly. Adding the transformation and inference time, HermiT outperforms E-KRHyper in all of the presented

benchmarks, but the difference is not that big. Taking into account that HermiT is a highly optimised reasoner for OWL ontologies and E-KRHyper is a versatile prover for FOL with certain extensions, this result is impressive.

As it was our intention to compare E-KRHyper with DL-provers, the typical ontologies we used tend to feature only *at-least* literals with low cardinalities. An unfortunate result of this is that the experimental optimisation to use DOI with the *at-least* rule has hardly any effect, as seen in the table. A more thorough evaluation of this particular optimisation would require tailor-made ontologies and must be regarded as future work.

5 Conclusion

E-KRHyper is a versatile automated theorem prover and model generator for first-order logic with equality. In version 1.4 it remains above average in the TPTP-benchmark and can compete with highly specialised and optimised DL provers. With the introduced extensions it gains access to new application domains. For problems that rely on the inequality of certain constants, we have shown that using DOI has significant benefits over explicit inequality-facts. The possibility to reason in $\mathcal{SHIQ}$ with overall good performance increases E-KRHyper's field of use and makes it even more flexible.

Acknowledgements. We would like to thank Sonja Polster, who provided us with the results of the DL-benchmarks.

References

1. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann Publishers Inc., San Francisco (1988)
2. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Trans. Comput. Log. (2009)
3. Baaz, M., Fermüller, C.G.: Resolution-based theorem proving for manyvalued logics. J. Symb. Comput. (1995)
4. Bachmair, L., Ganzinger, H.: Equational reasoning in saturation-based theorem proving. In: Bibel, W., Schmitt, P.H. (eds.) Automated Deduction — A Basis for Applications. Kluwer (1998)
5. Baumgartner, P., Burchardt, A.: Logic programming infrastructure for inferences on frameNet. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 591–603. Springer, Heidelberg (2004)
6. Baumgartner, P., Furbach, U.: Living books, automated deduction and other strange things. In: Hutter, D., Stephan, W. (eds.) Mechanizing Mathematical Reasoning. LNCS (LNAI), vol. 2605, pp. 249–267. Springer, Heidelberg (2005)
7. Baumgartner, P., Furbach, U., Gross-Hardt, M., Kleemann, T.: Model based deduction for database schema reasoning. In: Biundo, S., Frühwirth, T., Palm, G. (eds.) KI 2004. LNCS (LNAI), vol. 3238, pp. 168–182. Springer, Heidelberg (2004)
8. Baumgartner, P., Furbach, U., Gross-Hardt, M., Kleemann, T., Wernhard, C.: KRHyper inside - model based deduction in applications. Fachberichte informatik, Universität Koblenz-Landau (2003)

9. Baumgartner, P., Furbach, U., Groß-Hardt, M., Sinner, A.: Living book - deduction, slicing, and interaction. J. Autom. Reason (2004)
10. Baumgartner, P., Furbach, U., Koblenz, U.: PROTEIN: A PROver with a Theory Extension INterface. In: CADE-12, Proc. Springer, Heidelberg (1994)
11. Baumgartner, P., Furbach, U., Niemelä, I.: Hyper tableaux. In: Orłowska, E., Alferes, J.J., Moniz Pereira, L. (eds.) JELIA 1996. LNCS, vol. 1126, pp. 1–17. Springer, Heidelberg (1996)
12. Baumgartner, P., Furbach, U., Pelzer, B.: Hyper tableaux with equality. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 492–507. Springer, Heidelberg (2007)
13. Baumgartner, P., Mediratta, A.: Improving stable models-based planning by bidirectional search. In: International Conference on Knowledge Based Computer Systems (KBCS) (December 2004)
14. Baumgartner, P., Suchanek, F.M.: Automated Reasoning Support for First-Order Ontologies. In: Alferes, J.J., Bailey, J., May, W., Schwertel, U. (eds.) PPSWR 2006. LNCS, vol. 4187, pp. 18–32. Springer, Heidelberg (2006)
15. Bender, M.: E-hyper tableaux with distinct object identifiers. Arbeitsberichte aus dem Fachbereich Informatik 01/2013, Universität Koblenz-Landau (2013), http://www.uni-koblenz.de/FB4/Publications/Reports
16. Faßbender, D.: DLE-hyper tableau calculus. Diplomarbeit, University of Koblenz-Landau (February 2012)
17. Furbach, U., Glöckner, I., Helbig, H., Pelzer, B.: Loganswer - a deduction-based question answering system (System description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 139–146. Springer, Heidelberg (2008)
18. Furbach, U., Glöckner, I., Helbig, H., Pelzer, B.: Logic-based question answering. In: KI (2010)
19. Furbach, U., Glöckner, I., Pelzer, B.: An application of automated reasoning in natural language question answering. In: AI Commun. (2010) (PAAR Special Issue)
20. Ganzinger, H., Sofronie-Stokkermans, V.: Chaining techniques for automated theorem proving in many-valued logics. In: Proc. ISMVL 2000. IEEE Computer Society (2000)
21. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 299–314. Springer, Heidelberg (2011)
22. Horrocks, I.: Header Benchmarks (2009-2013), http://hermit-reasoner.com/2009/JAIR_benchmarks/ (accessed January 9, 2013)
23. Horrocks, I., Fensel, D., Broekstra, J., Decker, S., Erdmann, M., Goble, C., van Harmelen, F., Klein, M., Staab, S., Studer, R., Motta, E.: Oil: The ontology inference layer. Technical report, Vrije Universiteit Amsterdam, Faculty of Sciences (September 2000), http://www.ontoknowledge.org/oil/
24. INRIA. Objective Caml (OCaml) programming language website, http://ocaml.org/
25. McCune, W.: OTTER 3.3 Reference Manual. Argonne National Laboratory (2003)
26. Mossakowski, T., Maeder, C., Lüttich, K.: The heterogeneous tool set (hets). In: Beckert, B. (ed.) VERIFY. CEUR Workshop Proceedings, vol. 259. CEUR-WS.org (2007)
27. Motik, B., Shearer, R., Horrocks, I.: Optimized reasoning in description logics using hypertableaux. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 67–83. Springer, Heidelberg (2007)
28. Parsia, B., Sirin, E.: Pellet: An owl dl reasoner. In: 3rd International Semantic Web Conference (ISWC 2004) (2004)
29. Pelzer, B.: Project website of E-KRHyper (2007-2013), http://userp.uni-koblenz.de/~bpelzer/ekrhyper/ (accessed January 21, 2013)

30. Pelzer, B., Glöckner, I.: Combining theorem proving with natural language processing. In: Proceedings of the First International Workshop on Practical Aspects of Automated Reasoning (PAAR-2008/ESHOL-2008), Sydney, Australia, August 10-11. CEUR Workshop Proceedings (2008)
31. Pelzer, B., Wernhard, C.: System description: E- kRHyper. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 508–513. Springer, Heidelberg (2007)
32. Riazanov, A., Voronkov, A.: The design and implementation of vampire. AI Commun. (2002)
33. Schulz, S.: E - a brainiac theorem prover. AI Commun. (2002)
34. Schulz, S., Bonacina, M.P.: On handling distinct objects in the superposition calculus. In: Konev, B., Schulz, S. (eds.) Proc. IWIL (2005)
35. Schwitter, R.: Anaphora resolution involving interactive knowledge acquisition. In: Fuchs, N.E. (ed.) CNL 2009. LNCS, vol. 5972, pp. 36–55. Springer, Heidelberg (2010)
36. Sinner, A., Kleemann, T.: KRHyper - In Your Pocket. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 452–457. Springer, Heidelberg (2005)
37. Sutcliffe, G.: The TPTP website (2004-2013), `http://www.tptp.org` (accessed January 21, 2013)
38. Sutcliffe, G.: The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. J. Autom. Reason. (2009)
39. Wernhard, C.: System description: KRHyper. Fachberichte informatik, Universität Koblenz-Landau (2003)

Analysing Vote Counting Algorithms via Logic
And Its Application to the CADE Election Scheme

Bernhard Beckert[1], Rajeev Goré[2], and Carsten Schürmann[3]

[1] Karlsruhe Institute of Technology
beckert@kit.edu
[2] The Australian National University
Rajeev.Gore@anu.edu.au
[3] IT University of Copenhagen
carsten@itu.dk

Abstract. We present a method for using first-order logic to specify the semantics of preferences as used in common vote counting algorithms. We also present a corresponding system that uses Celf linear-logic programs to describe voting algorithms and which generates explicit examples when the algorithm departs from its specification. When we applied our method and system to analyse the vote counting algorithm used for electing the CADE Board of Trustees, we found that it strictly differs from the standard definition of Single Transferable Vote (STV). We therefore argue that "STV" is a misnomer for the CADE algorithm.

1 Introduction

Most research in electronic voting concerns correctness and voter-verifiability of vote-casting. It is also vital that vote counting and tabulation of election results enjoys the same trust as physical counting of paper ballots. Counting votes in a "first past the post" scheme is easy, but there are numerous preferential voting schemes for proportional representation where vote-counting is rather complex.

In social choice theory, the general problem of finding an election result that perfectly reflects the electorate's collective preferences has no solution. Collective preferences can even be cyclic (Condorcet's paradox). Various voting schemes (and corresponding algorithms) exist that attempt to provide "good" election results from the preferences expressed in voters' ballots. They compromise in different ways, for example w.r.t. majority rule vs. minority protection.

In preferential voting, the correct election result for a given set of votes is usually defined algorithmically (in natural language or in pseudo-code) rather than as clear, concise, formal and declarative specifications. A prominent class of such algorithms is the Single Transferable Vote (STV) scheme (see Section 2). Variants of STV are used in many jurisdictions around the world. Testing and verification methods are useful for validating that an implementation refines a high-level algorithmic description, but without good declarative specifications they remain ineffective to gain deeper insight into the democratic nature of the scheme. Unfortunately, the complexity of STV makes it difficult to render its declarative properties succinctly: a possible solution is presented in Section 3.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 135–144, 2013.

In Section 4, we then describe an automated reasoning system to check vote counting schemes w.r.t. their declarative properties. Our technique applies to STV but can be generalised to other schemes as well. Building on earlier work [4], we use the concurrent logical framework Celf [8] for (1) representing vote counting algorithms, (2) representing declarative properties, and (3) for building tools such as a bounded model checker for constructing counter examples. The Celf source code can be found at `http://www.demtech.dk/wiki/CADE24-STV`.

We applied our methodology to the "STV" vote-counting algorithm used to elect the CADE Board of Trustees (Section 5) and found that this algorithm does not implement a proportional voting scheme. Thus it is a misnomer to call the CADE algorithm a scheme for Single Transferable Voting (Section 6).

Related work at the Australian National University has analysed a commercial implementation of the Hare-Clark (STV) scheme used in Australian Capital Territory parliamentary elections [6] while Cochran has formally specified and analysed the Danish and Irish vote counting algorithms [3]. Both concentrate on verifying implementations.We concentrate on analysing abstract algorithms.

2 The Single Transferable Vote Scheme

Single transferable vote (STV) is a preferential voting scheme [9] for multi-member constituencies aiming to achieve proportional representation according to the voters' preferences. Suppose that $\mathtt{C}$ candidates, numbered $1, 2, \ldots, \mathtt{C}$, are competing for $\mathtt{S} > 0$ vacant seats in an election. Furthermore, assume that $\mathtt{V} \geq 0$ votes have been cast and are collected in a ballot box. It is commonly agreed that for $k \leq \mathtt{C}$, a vote $[P_1, P_2, \ldots, P_k]$ ranks a subset of the candidates with $P_i \in \{1, 2, \ldots, \mathtt{C}\}$ in decreasing order of preference. Each vote defines a partial order on candidates. STV first computes a quota necessary to obtain a seat. Different definitions of quotas are used in practice and the most common is the Droop quota $\mathtt{Q} = \lfloor \mathtt{V}/(\mathtt{S}+1) \rfloor + 1$. Then, STV computes the result using an iterative process, which repeats the following two steps until either a winner is found for every seat or no further candidate can be elected:

1. Any candidate with $\mathtt{Q}$ or more first-preference votes is declared elected. The $\mathtt{Q}$ votes used to elect such a candidate are removed from the ballot box. If the elected candidate has more votes than the quota $\mathtt{Q}$, these surplus votes are transferred to the next candidates according to preference.
2. If no candidate reaches the quota, the candidate with the fewest first-preference votes is eliminated and that candidate's votes are transferred in the same way as described in step 1.

Example 1. Assume there are four candidates A, B, C, D for two vacant seats, and the votes to be counted are $[A, B, D], [A, B, D], [A, B, D], [D, C], [C, D]$. The Droop quota here is $\mathtt{Q} = \lfloor 5/(2+1) \rfloor + 1 = 2$. In the first iteration, we tally first preferences, only A meets the quota and is hence elected. Two votes $[A, B, D]$ are deleted, the third is a surplus vote. It is transferred and transformed into $[B, D]$. In the second iteration, no candidate reaches the quota, thus the weakest of

the remaining candidates B, C, D is eliminated – which one depends on the kind of tie-breaker used as all three have exactly one first-preference vote at that point. (1) If the tie-break eliminates B, the aforementioned transferred vote $[B, D]$ will be transferred again and will become a vote for D, so that D will be elected in the next iteration. (2) If the tie-break eliminates C, the vote $[C, D]$ will be transferred into a vote for D, and thus D will be elected. (3) If the tie-break eliminates D, then C will be elected, analogously, in the next iteration. In summary, the algorithm reports either $[A, D]$ or $[A, C]$ as the election result but not, for example, $[A, B]$ or $[B, D]$. If the number of second-preference votes is used as a tie-breaker, then B is eliminated first (case 1 above).

This illustrates that STV, as given above in informal English, defines an entire family of vote counting algorithms. There are a number of parameters to play with, which we name for later reference: the choice of quota (QUOTA/DROOP, QUOTA/HARE, QUOTA/MAJORITY), the choice of tie-breaker (TIE), the possible resurrection of already eliminated candidates when they receive transferred votes in later rounds (ZOMBIE), the automatic placement of candidates once there are as many vacant seats as remaining candidates (AUTOFILL).

As we argue in Section 5, further options giving schemes that cannot be considered part of the STV family are the persistence of votes used in electing candidates from one iteration to the next (NODEL), or restarting with the original ballot box (RESTART) after a candidate has been elected (with the elected candidate removed). RESTART is a combination of NODEL and ZOMBIE.

3 Semantic Criteria for Judging Voting Schemes

Voting schemes are frequently under-specified via something like "The members of parliament must be elected by a regular, direct, and secret election", or over-specified via a concrete (pseudo-code) algorithm in the law text that may itself contain bugs. So what is a good specification of STV?

Many semantic criteria have been proposed for preferential vote-counting algorithms including the majority criterion, Condorcet criterion, monotonicity criterion, to name a few [1]. The majority criterion says that *if one candidate is highest ranked by a majority of voters, then that candidate must be elected.* A violation of the majority criterion is clearly undemocratic. But to analyse and distinguish variants of (democratic) voting schemes, stronger criteria are needed that are tailored for the particular scheme. For some voting schemes, axiomatic criteria can fully characterise the correct election result (e.g. the Borda Rule [5]).

However, for STV schemes, writing a declarative specification that fully characterises the election result is hard. For our analysis of STV, we have instead devised several semantic criteria that capture and approximate the essence of STV and are suitable for program verification. We mention only two, but remark that for a more thorough analysis more such criteria will have to be added.

(1): There must be enough votes for each elected candidate.
(2): If the preferences of *all* voters w.r.t. two particular candidates are consistent, then that collective preference is not contradicted by the election result.[1]

The first criterion only considers number of votes and ignores preferences, while the second criterion only considers preferences and ignores number of votes. This separation of the two dimensions (number of votes and preferences) is the key to finding criteria that can be described declaratively. Due to space restrictions, we do not further consider Criterion 2 in this paper but concentrate on Criterion 1.

The first criterion is only justified for versions of STV that do not use AUTOFILL. It captures the fact that the ballot box can be partitioned in such a way that each partition justifies the seating of a candidate: Each class contains quota `Q` ballots, each of which mentions the candidate elect at least once.

Example 2. Returning to Example 1, note that the election result $[A, D]$ satisfies Criterion 1 with the partition $\{[A, B, D], [A, B, D]\}, \{[C, D], [D, C]\}, \{[A, B, D]\}$. The incorrect election result $[B, D]$ also satisfies Criterion 1 choosing the same partition (because the ordering of A and B is not considered), which shows that the criterion is not a complete specification of the election result. But, the incorrect result $[A, B]$ is not supported by this or any other partition.

We use first-order logic over the theories of natural numbers and arrays with the following notation in addition to the notation defined previously:

b: is the ballot box, where $b[i, j]$ is the number of the candidate that is ranked by voter i in the jth place. If the voter does not rank all candidates, then $b[i, j] = 0$ for the empty places.
r: is the result, where $r[i]$ is the ith candidate that is elected ($1 \leq i \leq \mathtt{S}$). If less than `S` candidates are elected, then $r[i] = 0$ for the empty seats.

We also use an existentially quantified array a that represents the partition and the assignment of classes in the partition to elected candidates as follows:

$a[i] = k$ if the ith vote supports the kth elected candidate $r[k]$. If the ith vote does not support any elected candidate, then $a[i] = 0$.

Then, the formula $\phi = \exists a(\phi_1 \wedge \ldots \wedge \phi_4)$ is the existentially quantified conjunction:

$$\forall i\big(1 \leq i \leq \mathtt{V} \rightarrow 0 \leq a[i] \leq \mathtt{S}\big) \qquad (\phi_1)$$

$$\forall i\big(1 \leq i \leq \mathtt{V} \rightarrow (a[i] \neq 0 \rightarrow r[a[i]] \neq 0\big) \qquad (\phi_2)$$

[1] This criterion is weaker than what is known as the Condorcet criterion. We assume a preference to be collective if *all* voters agree (or at least not disagree), while the Condorcet criterion assumes a preference to be collective if it is supported by a *majority* of voters. It is well known that the (stronger) Condorcet criterion is not satisfied by standard STV, so it is not suitable for our purposes.

$$\forall i\big((1 \leq i \leq \mathtt{V} \wedge a[i] \neq 0) \rightarrow \exists j(1 \leq j \leq \mathtt{C} \wedge b[i,j] = r[a[i]])\big) \qquad (\phi_3)$$

$$\begin{aligned}&\forall k\big((1 \leq k \leq \mathtt{S} \wedge r[k] \neq 0) \rightarrow \qquad (\phi_4)\\ &\quad \exists count(count[0] = 0 \wedge\\ &\qquad \forall i(1 \leq i \leq \mathtt{V} \rightarrow (a[i] = k \rightarrow count[i] = count[i-1] + 1) \wedge\\ &\qquad\qquad (a[i] \neq k \rightarrow count[i] = count[i-1])) \wedge\\ &\qquad count[\mathtt{V}] = \mathtt{Q})\big)\end{aligned}$$

Formulas ϕ_1 and ϕ_2 express well-formedness of the partition. Formula ϕ_3 expresses that only votes can support a candidate in which that candidate is somewhere ranked. Formula ϕ_4 expresses that each class supporting a particular elected candidate has exactly `Q` elements. To formalise this, we use an array *count* such that $count[i]$ is the number of supporters among votes $1, \ldots, i$ that support the kth elected candidate.

4 The System for Analysing Vote Counting Algorithms

Votes are resources that must be counted exactly once, which already suggests that linear logics are well-suited for representing voting-algorithms (see Section 2 and Criterion 1 in Section 3). We therefore used the (linear) logical framework Celf [8]. We considered using model checkers and SMT solvers for the bounded model-checking part, but we stayed with Celf, mainly as a matter of convenience. Because of space restrictions, we only sketch the system here.

4.1 Vote Counting Algorithms as Linear-Logic Programs

The Celf logical framework is based on intuitionistic linear logic. Its operational semantics is proof search, which means that running a vote counting algorithm is tantamount to constructing a *derivation* for "$\Gamma; \Delta \vdash$ `run` i `C V S` w d". We explain the symbols in turn. The Γ is the unrestricted (intuitionistic) context. Its declarations, like AUTOFILL, NODEL, etc. define precisely the particular STV algorithm to be analysed. During execution, Γ is also populated with assumptions about who was elected and who was defeated.

The Δ to the right of Γ denotes the linear context that contains assumptions that must be used exactly once. It contains, for example, all information about the ballot-box, running totals, etc. The ballot box is represented by a multi-set of assumptions `uncounted-ballot` A L (first preference A, remaining preferences L); the running totals as a multi-set of assumptions `hopeful` A N, summarising that A's running total is N.

`run` is a 6-ary predicate, relating the number of times i that STV may be restarted (see RESTART), the number of candidates `C`, the number of cast votes `V`, the number of seats `S` that should be filled with each iteration, the list of winners w, and the list of defeated candidates d. The total number of seats filled by the algorithm is hence $i \times \mathtt{S}$.

To save space, we present only one of the rules implementing STV. An introduction on how to represent STV counting algorithms in Celf is given in [4].

This rule elects candidate A after receiving an additional vote that meets the quota.

$$\begin{aligned}&\texttt{count/2} : \texttt{count-ballots}\ (S+2)\ (H+1)\ (U+1) \otimes \\ &\qquad \texttt{uncounted-ballot}\ A\ L \otimes \\ &\qquad \texttt{hopeful}\ A\ N \otimes\ !\texttt{quota}\ Q \otimes\ !\texttt{nat-lesseq}\ Q\ (N+1) \otimes \\ &\qquad \texttt{winners}\ W\ D \\ &\qquad\quad \multimap \{\texttt{counted-ballot}\ A\ L \otimes\ !\texttt{elected}\ A \otimes \\ &\qquad\qquad \texttt{winners (scons}\ A\ (N+1)\ W)\ D \otimes \\ &\qquad\qquad \texttt{count-ballots}\ (S+1)\ H\ U\}.\end{aligned}$$

All uppercase variables are universally bound, we write $\multimap$ for linear implication, $\otimes$ for multiplicative conjunction, ! for the bang modality permitting unrestricted assumptions to appear in declarations, and {...} for the polarity shift from positive to negative formulas (as Celf implements a focused linear logic). The rule `count/2` can be read as a forward-chaining multi-set rewrite rule. When no candidate reaches the !`quota` Q, the candidate with the fewest votes must be eliminated, and its (already counted) votes redistributed. The bang in front of `quota` indicates that this is an unrestricted assumption that should not be consumed. It is therefore mandatory to keep information about `counted` ballots around, and we do this by replacing an `uncounted-ballot` A L by a `counted-ballot` A L.

Theorem 1 (Standard STV). *Let* $\Gamma =$ QUOTA/DROOP, AUTOFILL, TIE*, and let* $\Delta =$ ballot box + initialized running counts*, then* `run` 1 `C V S` w d *is provable if and only if* w *corresponds to the list of candidates elected by the standard STV algorithm, and* d *is the list of defeated candidates.*

4.2 Bounded Model Checker

Our bounded model checker is implemented in Celf, taking advantage of the *generate* and *test* behaviour of logic programs. Our system provides an implementation of Criterion 1 as a linear logic program: `sem` W D. Other criteria may be implemented analogously. The model checker generates all possible ballot boxes up to a given bound. The bound comprises the maximal number of permitted RESTARTs max_i, the maximal number of candidates $max_{\mathtt{C}}$, the maximal size of the ballot box $\max_{\mathtt{V}}$, and the maximal number of seats $max_{\mathtt{S}}$. Checking the algorithm for a particular input corresponds to running the query:

$$\Gamma; \Delta \vdash (\texttt{run}\ i\ c\ v\ s\ W\ D)\ \&\ (\texttt{sem}\ W\ D)$$

As above, Γ selects the algorithm for the desired version of STV, Δ, i, c, v, s are inputs for the algorithm that have been generated by the model checker. We use additive conjunction & in a clever way: it copies the ballot box and allows the box to be used both for running STV and for semantically checking the result.

As inherent in *bounded* model checking, we get a semi-decision procedure. The analysis terminates for any given bound (this is easy to prove by an inspection of the linear logic program). But only in the negative case, where we get a counter

example, the model checker provides a definite answer to the question of whether the algorithm always computes an election result satisfying the given criterion.

In the positive case, where the model checker constructs as many solutions to the above query as there are ballot boxes, we can conclude that the particular STV algorithm selected by Γ computes valid solutions for all possible elections up to the given bound. Note, that this conclusion requires the vote counting to be deterministic because the current version of the checker does not backtrack over different results for the same input and only validates the first election result found for a particular ballot box. This is not a critical limitation in practice. Although various ways of resolving non-determinism in STV exist, it is important to clearly specify how it is resolved in real-world implementations of TIE. Otherwise, choices by the election officials (or their computers) during counting could influence the result, which is clearly undesirable. Our checker could backtrack over election results but would require greater runtime.

The more interesting case is when the model checker fails to find a solution for one of two reasons. Either, the STV algorithm did not manage to construct an election result for some ballot box, a case that may happen, for example, if AUTOFILL is not selected. Or the model checker was unable to justify an election result w.r.t. the semantic criteria (the really negative case). The Celf tracing model provides enough information to deduce where to find the culprit.

5 Case Study: CADE-STV

The bylaws of the Conference on Automated Deduction (CADE) give an algorithm for counting the ballots for electing the Board of Trustees [2] which is identical to Figure 1 except for typesetting. It has been implemented and used in several elections for the CADE Board of Trustees. It has also been used several times for the election of members to the TABLEAUX Steering Committee.

Note that the specification of CADE-STV is not formal. Although the pseudo code language that is used may be intuitive for programmers, it does not come with a precise operational semantics. Despite being semi-formal, the pseudo code lacks precision in how to break a tie when eliminating or seating candidates.

The CADE-STV algorithm deviates from standard STV in using RESTART, which combines NODEL and ZOMBIE. Moreover, QUOTA/MAJORITY is used instead of QUOTA/DROOP and there is no AUTOFILL, which is unusual. Presumably, QUOTA/MAJORITY was introduced into CADE-STV following criticism of DROOP/QUOTA by David Plaisted [7].

Example 3. Let us run CADE-STV on Example 1. First, we compute the majority quota $\mathtt{Q} = 3$. In the first iteration, A has three first preferences, which means that A is the majority winner and is seated. Since CADE-STV uses RESTART, A's votes are not deleted but are redistributed at the end of the first iteration. Now the ballot box contains $[B, D], [B, D], [B, D], [D, C], [C, D]$. Following the algorithm, we observe that now B is the majority candidate with 3 first preference votes and is seated. The election is over, and the election result is $[A, B]$.

Problem: `M` voters must elect `K` of `N` candidates.
Input: $\mathtt{M} \times \mathtt{N}$ matrix, `Tbl` of votes.
`Tbl(i, j)` $= \mathtt{p}$, $1 \leq \mathtt{p} \leq \mathtt{N}$, means that voter `i` gave preference `p` to candidate `j`. Every voter can support `n` ($1 \leq \mathtt{n} \leq \mathtt{N}$) of the `N` candidates, and has to give a preference order between those `n` candidates. This is expressed by assigning a preference between 1 (highest preference) to `n` (lowest preference) to each of the supported candidates. Each of the values $1 \ldots \mathtt{n}$ is assigned to exactly one candidate. All candidates not supported receive a preference of $\mathtt{N} + 1$.
Weakest candidate: The candidate with the fewest votes of preference 1. Ties are broken by fewest votes of preference 2, then 3, etc.
Equally weak candidates: `c` is equally weak as `w` iff `c` and `w` have the same number of votes of preference 1, 2, etc.
Output: List of `K` elected candidates in order of election.

```
Redistribute(k, Tbl):
    for v <-- 1 to M
        p <-- Tbl(v,k)  {* v's preference for candidate k *}
        for c <-- 1 to N
            p' <-- Tbl(v,c) {* v's preference for candidate c *}
            if p' > p and not p' == N+1 then
                decrement Tbl(v,c) by one
        end for
    end for
    Now remove candidate k from Tbl  {* column k *}

Procedure STV
    Elected <-- empty
    T <-- Tbl               {* Start with the original vote matrix *}
    for E <-- 1 to K
        N' <-- N-E+1   {* Choose a winner among N' candidates *}
        T' <-- T       {* store the current vote matrix *}
        while (no candidate has a majority of 1st preferences)
            w <-- one weakest candidate
            for all candidates c {* remove all weakest candidates *}
                if c is equally weak as w
                    Redistribute(c,T)
            end for
        end while
        win <-- the majority candidate
        Elected <-- append(Elected, [win])
        T <-- T'       {* restore back to N' candidates *}
        Redistribute(win, T)  {* remove winner & redistrb. votes *}
    end for
End STV
```

Fig. 1. Description of the CADE-STV algorithm [2]

Standard STV and CADE-STV produce different results on the same votes. Example 2 has already shown that $[A, B]$ is "incorrect" as it violates Criterion 1.

Theorem 2. *Let* $\Gamma = \text{QUOTA/MAJORITY}, \text{RESTART}$ *and* $\Delta =$ ballot box + initialized running counts*, then* `run S C V` 1 w d *is provable if and only if* w *is the list of candidates elected by CADE-STV, and* d *is the list of defeated candidates.*

Running the bounded model checker confirms that the election results computed by CADE-STV do not always satisfy Criterion 1. Indeed, it finds smaller counter examples than our running example, but these are not as illustrative.

The effect of the differences between standard STV and CADE-STV is clarified by the following theorem and its corollary: in certain cases, there is no proportional representation in the election results computed by CADE-STV.

Theorem 3. *If a majority of voters all vote identically with* $[P_1, \ldots, P_k]$*, then CADE-STV will elect exactly these candidates in exactly this order.*

Proof. Since a majority of voters choose P_1 as their first preference, no other candidate can meet the "majority quota". Thus P_1 is elected in the first round. When redistributing the ballots, each of the majority of ballots with P_1 as first preference have P_2 as second preference. All become first preferences for P_2. Thus candidate P_2 is guaranteed to have a majority of first preferences and is elected in round two, and so on until all vacancies are filled. □

Corollary 1. *If the electorate consists of two diametrically opposed camps that vote for their candidates only, in some fixed order, then all candidates from the majority camp are elected and no candidates from the minority camp are elected.*

Standard STV does not use RESTART (nor NODEL), so it elects the first ranked candidate of the majority. It then distributes only the surplus votes, not all votes as done by CADE-STV. Thus the second preference from the majority is not necessarily elected next. Majorities do not rule outright in standard STV.

6 Conclusion

The experiments with our tool show that CADE's "single transferable vote" voting scheme does not satisfy the intuitive semantic criterion defined in Section 3 and does not achieve proportional representation. We suspect that CADE's voting scheme was intended to combine the advantages of preferential and majority voting (nothing can happen that the majority does not want). Unfortunately, it also combines their disadvantages (no proportional representation).

Our observations do not imply that CADE voting is undemocratic. But calling the CADE algorithm "Single Transferable Vote" is a misnomer because the goal of proportional representation is inherent to STV. The CADE algorithm is actually closer to what is known as *Majority Preference Voting*.

CADE-STV has been used for many years. It has been implemented, tested, re-implemented, and re-tested by various people. Its properties have been thoroughly discussed at various times by the CADE Trustees. But to our knowledge, the problems outlined in this paper have not been observed before, which clearly indicates that a formal analysis like the one presented here is indispensable.

References

1. Brandt, F., Conitzer, V., Endriss, U.: Computational social choice. In: Weiss, G. (ed.) Multiagent Systems. MIT Press (2012)
2. CADE Inc.: CADE Bylaws (effective November 1, 1996; amended July/August 2000), `http://www.cadeinc.org/Bylaws.html` (accessed January 20, 2013)

3. Cochran, D.: Formal Specification and Analysis of Danish and Irish Ballot Counting Algorithms. Ph.D. thesis, IT University of Copenhagen (2012)
4. DeYoung, H., Schürmann, C.: Linear logical voting protocols. In: Kiayias, A., Lipmaa, H. (eds.) VoteID 2011. LNCS, vol. 7187, pp. 53–70. Springer, Heidelberg (2012)
5. Farkas, D., Nitzan, S.: The Borda Rule and Pareto Stability: A comment. Econometrica 47(5), 1305–1306 (1979)
6. Logic and Computation Group at ANU: Formal methods applied to electronic voting systems, `http://users.cecs.anu.edu.au/~rpg/EVoting/index.html` (retrieved) (accessed January 20, 2013)
7. Plaisted, D.A.: A consideration of the new CADE bylaws, `http://www.cs.unc.edu/Research/mi/consideration.html` (accessed January 20, 2013)
8. Schack-Nielsen, A., Schürmann, C.: Celf: A logical framework for deductive and concurrent systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 320–326. Springer, Heidelberg (2008)
9. Wikipedia: Single transferable vote, `http://en.wikipedia.org/wiki/Single_transferable_vote` (accessed January 20, 2013)

Automated Reasoning, Fast and Slow*

Natarajan Shankar

Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA
shankar@csl.sri.com

Abstract. Psychologists have argued that human behavior is the result of the interaction between two different cognitive modules. System 1 is fast, intuitive, and error-prone, whereas System 2 is slow, logical, and reliable. When it comes to reasoning, the field of automated deduction has focused its attention on the slow System 2 processes. We argue that there is an increasing role for forms of reasoning that are closer to System 1 particularly in tasks that involve uncertainty, partial knowledge, and discrimination. The interaction between these two systems of reasoning is also fertile ground for further exploration. We present some tentative and preliminary speculation on the prospects for automated reasoning in the style of System 1, and the synergy with the more traditional System 2 strand of work. We explore this interaction by focusing on the use of cues in perception, reasoning, and communication.

To be useful, a memory has to be recalled. Memory retrieval depends on the presence of appropriate cues that an animal can associate with its learning experiences. The cues can be external, such as a sensory stimulus in habituation, sensitization, and classical conditioning, or internal, sparked by an idea or an urge.

Eric Kandel, *In Search of Memory* [18]

The situation has provided a cue; this cue has given the expert access to information stored in memory, and the information provides the answer. Intuition is nothing more and nothing less than recognition.

Herbert Simon [38]

Over the last six decades, the field of *Automated Reasoning* has focused single-mindedly on a specific form of reasoning, namely, logico-deductive inference. Work on deductive inference has yielded significant progress, particularly in the form of robust proof search methods, efficient decision procedures, and expressive logics for proof checking. This progress has led to the modeling and verification

* This work was supported by NSF Grant CSR-EHCS(CPS)-0834810, NASA Cooperative Agreement NNA10DE73C and by DARPA under agreement number FA8750-12-C-0284. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. Sam Owre and Shalini Ghosh provided insightful feedback on earlier drafts of the paper.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 145–161, 2013.

of complex hardware and software systems [23,3], and the formalization of large tracts of mathematical knowledge [13]. The field can certainly take pride in the progress that it has made in the automation of deductive inference, and this line of work certainly appears to have a bright future. However, logical deduction is much too rigid and brittle for many tasks that call for mechanized reasoning. For example, the availability of large volumes of data call for efficient forms of *inductive* reasoning to extract meaningful rules from the data, and *abductive* reasoning to infer hidden information from visible observations. In many applications, speed is essential, so that the cost of inference must be seen as a factor. We present a few modest examples that motivate the diversification of automated reasoning beyond deduction. More specifically, we outline a framework we call *cueology* (pronounced "Q-ology") for fast, approximate inference for discriminative tasks based on cues.

First, a little detour through cognitive science as an explanation for the title. In his 2012 book, *Thinking, Fast and Slow* [17], Daniel Kahneman surveys the research on psychology, focusing on the interaction between two systems of cognition. Cognitive psychologists have studied human reasoning to examine the gap between "rational" modes of reasoning and those typically used by humans. They have identified a *dual-process* architecture of cognition in the human mind that combines two distinct systems. The first, System 1, is involuntary, parallel, intuitive, and fallible, whereas the second, System 2, is effortful, sequential, deliberative, and mostly reliable. As an example of a System 1 activity would be to "drive a car on an empty road" in contrast to a System 2 activity such as to "check the validity of a complex argument." System 2 keeps a watchful eye on System 1, and is activated either when System 1 perceives something surprising or when a task is too difficult for it. However, System 2 is lazy, and the combination of the two systems can often lead to errors. Sometimes, this is because System 2 is not engaged, and in other cases, it is either preoccupied or exhausted. As an example of the former, students are given the puzzle: *A bat and ball cost $1.10. The bat costs one dollar more than the ball. How much does the ball cost?* More than half the students respond with the incorrect answer: 10 cents. As an example of System 2 being preoccupied, a famous video [1] by Chabris and Simons shows a basketball game between a team of players dressed in white against another team dressed in black. It sets the viewer the tasks of counting the number of passes between players from the white team. When the subject is engaged in this intense System 2 activity, about halfway through the video, a woman in a gorilla suite walks into the frame, thumps her chest and walks across the court. This is typically missed by the subject engaged in the counting task, a phenomenon called inattentional blindness.

Kahneman describes his various other ways in which the interaction of the two systems of cognition leads to a number of fallacies, several of which were identified in collaboration with Amos Tversky. Priming and repetition can easily bias thought. For example, polling stations located near a school influence voters to vote in favor of pro-school ballot propositions. They are swayed by

[1] `http://www.theinvisiblegorilla.com/gorilla_experiment.html`

small sample sizes, and are not particularly consistent when it comes to calculating probabilities and risks. They are also prone to logical fallacies, as with the conjunction fallacy. Here, subjects are told that *Linda is thirty-one years old, single, outspoken, and very bright. She majored in philosophy. As a student, she was deeply concerned with issues of discrimination and social justice, and also participated in antinuclear demonstrations.* Is it more likely that she is a bank teller or a bank teller who is active in the feminist movement? A substantial majority of those tested pick the latter, even though the probability that Linda is a bank teller and an active feminist cannot exceed the probability that Linda is a bank teller.

Kahneman and Tversky have their critics, most prominently the psychologist Gerd Gigerenzer [10] who argues that many of these fallacies are not flaws but the result of Herbert Simon's notion of bounded rationality. Most human reasoning takes place in contexts where there is limited information and time. Gigerenzer and his colleagues argue that *fast and frugal* heuristics, typically simple decision trees based on a small subset of cues, are often sufficiently effective compared to more complex decision structures that take some weighted combination of cues. For example, a *recognition heuristic* can be used to construct a stock portfolio that consistently beats the market by picking companies that are recognized by a random sampling of people

Regardless of whether human cognition is a good model for automated reasoning, fields like cognitive psychology and probability theory offer a science of rational reasoning that can serve as a foundation for both human and machine cognition. The main challenge is to identify forms of reasoning that are rationally justifiable, particularly in data-intensive contexts such as science, gambling, or sports, where noise and uncertainty abound, and yet rationality is a clear objective. The dual-process model of mind is appropriate for extracting the signal from the overwhelming quantity of sensory data. A good deal of the System 2 reasoning can be carried out off-line to identify suitable cues that are embedded in System 1 processes that continue to be monitored by other System 2 processes. Such a dual-process model is also applicable to traditional applications of automated reasoning where quick and unreliable inference heuristics are applied first, and the more reliable but computationally intensive inference mechanisms are only triggered when the easy heuristics fail.

We offer a very preliminary outline of a science of fast reasoning based on cues that we term cueology. Section 1 briefly summarizes the common criticisms against deduction. Section 2 gives a quick overview of probabilistic reasoning using generative models and its relation to logical inference. Section 3 examines the use of cues for fast and approximate prediction or discrimination tasks. Section 4 discusses the synergy between fast and slow inference techniques in automated reasoning, and the conclusions are presented in Section 5.

1 A Critique of Deductive Inference

We can regard perception, memory and induction as the three fundamental ways of acquiring knowledge; deduction on the other hand is merely

a method of arranging our knowledge and eliminating inconsistencies or contradictions.

F. P. Ramsey[32]

There is no question that logical deduction has been unreasonably effective both at the foundational level of identifying the valid modes of reasoning, and at the practical level of formalizing knowledge and automating reasoning. Logical precision and consistency are central to deduction: a claim A follows from a knowledge base K exactly when $K, \neg A$ is inconsistent. Recent progress in mechanized inference has substantially reduced the level of effort and difficulty in formally verifying mathematical knowledge using interactive proof checkers. SAT and SMT solvers are being used in a number of inferential tasks from ranging from AI planning to hardware and software design and verification. However, there is still a level of brittleness to deductive inference that makes it inappropriate for many tasks that do call for some form of inference. For example, data frequently contain errors and noise. Inference based on such data needs to be resilient to these errors. In many applications, speed is of the essence and can be traded off for precision. Techniques from automated deduction can be adapted for these applications, and there is already a considerable body of work in this direction [8].

A number of criticisms have been levelled against the use of logic as a framework for knowledge representation and reasoning. The basic objections are

1. Logical representations entail uniform methods of inference, where problem-specific algorithms and representations can be more effective.
2. Logic is monotonic, so that no default inferences can be made in the absence of knowledge.
3. Logic does not cope well with uncertainty, whereas knowledge in the real-world is often fragmentary and unreliable.

These critiques actually highlight some of the strengths of logic. It serves as a convenient interface so that inference mechanisms can be developed and optimized for entire classes of models. Conversely, models can be developed without being overly sensitive to the actual inference mechanisms. The success of SAT solving illustrates this: it is not easy to develop bespoke algorithmic solutions for many of the inference problems where generic SAT solvers are already quite effective. The monotonicity of logic allows reasoning to be decomposed and factored into theories and lemmas. The idealized and unambiguous nature of logic is what makes it so effective for building models and for identifying errors of representation or reasoning.

By leveraging abstraction and modularity, inference algorithms have been successful across a wide range of fields including databases, programming languages, hardware and software verification, and symbolic systems biology. The success of automated deduction can be extended to a range of other inference tasks that do not require the same level of precision. Examples of such tasks include expert systems, natural language processing, social network analysis, robotics, and predictive analytics. Some of the benefits of this diversification can accrue

to automated reasoning itself, for example, in learning, adapting, and matching inference techniques to new challenges.

In the next section, we present probabilistic inference as a step in this direction.

2 Probabilistic Inference

> Our theme is simply: *Probability Theory as Extended Logic.* ... the mathematical rules of probability theory are not merely rules for calculating frequencies of "random variables"; they are also the unique consistent rules for conducting inference (*i.e.*, plausible reasoning) of any kind ...
>
> E. T. Jaynes, *Probability Theory: The Logic of Science* [15]

An acquaintance tells you she has two children, one is a boy born on Tuesday. What is the probability she has two boys? It is tempting to think that the reference to Tuesday is irrelevant, and that the probability of the other child being a boy is 1/2. There are two possibilities for the gender of each child, and seven possibilities for the day of the week for the birthday. If we restrict ourselves to the possibilities where at least one child is a boy born on a Tuesday, then either the first child is a boy born on a Tuesday, and there are 14 possibilities for the second child including 7 instances where the second child is a boy. Otherwise, if the first child is not a boy born on a Tuesday but the second one is, and there are 13 possibilities for the first child including 6 instances where the first child is a boy. The probability is therefore 13/27.

Probabilistic inference as illustrated by this example is at the foundation of a number of disciplines from information theory [24] to psephology (the study of polling and elections) [37]. Applications of probabilistic inference range from robotics to natural language understanding. On the one hand, probability is just another mathematical theory that is easily formalized in interactive proof checkers, and such formalizations exist in Coq [2], and PVS [27]. On the other hand, the practical applications of probabilistic inference employ specialized inference mechanisms that are quite different from those of logical deduction. We give a brief introduction to a few of the approaches to probabilistic inference. We then present Markov Logic developed by Pedro Domingos and his colleagues [8] as a combination of probabilistic and logical inference. We briefly describe the MC-SAT inference algorithm [31] that is implemented in systems such as Alchemy (`http://code.google.com/p/alchemy-2/`) and the Probabilistic Consistency Engine (PCE; developed jointly with Sam Owre).

A probabability space is a triple $\langle \Omega, \mathcal{F}, P \rangle$ consisting of

1. A *sample space* Ω which is a set of outcomes. We restrict our attention to finite sample spaces of the form $\{e_1, \ldots, e_n\}$.
2. An event algebra $\mathcal{F}$, a nonempty set of subsets of Ω closed under countable unions and complements.
3. A probability measure P mapping events to probabilities so that $P(E) \geq 0$ for all $E \in \mathcal{F}$, $P(\Omega) = 1$, and $P(\bigcup_i E_i) = \Sigma_i P(E_i)$, where $\langle E_0, E_1, \ldots \rangle$ is a sequence of pairwise disjoint events.

Example 1. For a fair 6-sided dice, the probability $P(i)$ for $1 \leq i \leq 6$ is $\frac{1}{6}$, and the probability of an even number, i.e., the event $\{2, 4, 6\}$ is $\frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$

The conditional probability of one event A relative to another one B is given $P(A \cap B)/P(B)$. Bayes' theorem relates the conditional and marginal probabilities of events A and B, where $P(B)$ is nonzero.

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}.$$

The significance of Bayes theorem is that if M is a model and D is an observation, the *posterior probability* of a model M explaining the observation D, i.e., $P(M|D)$ can be computed in terms of the *prior probability* $P(M)$ of the model M the probability $P(D|M)$ of the observation given the model, and the *marginal probability* probability of the observation. For example, if there are three competing models M_1, M_2, M_3, then we can find use Bayes theorem to identify the model that best explains the observation.

Medical diagnosis offers a simple example of Bayesian reasoning as illustrated by the following example from the Wikipedia page on *Bayesian inference*. We have a test for a disease that returns positive or negative results. If the patient has the disease, the test is positive with probability .99. If the patient does not have the disease, the test is positive with probability .05. A patient has the disease with probability .001. *What is the probability that a patient with a positive test has the disease?* Abbreviating the event corresponding to the presence of the disease as D, and the event corresponding to a positive test result as R, this can be calculated as

$$P(D|R) = P(R|D)P(D)/P(R) = .99 \times .001/(.99 \times .001 + .05 \times .999) = 0.0194$$

This example illustrates why it is important to pay attention to the base rate probability of the disease.

Two events E_1 and E_2 are said to be *conditionally independent* under a third event C if $P(E_1 \cap E_2|C) = P(E|C)P(F|C)$. Conditional independence can be exploited to construct compact probabilistic models.

For a sample space Ω, a random variable X ranging over some value space V is a mapping from Ω to V. For $x \in V$, the probability mass function $P(X = x)$ is computed as $P(E)$, where $E = \{e \in \Omega | X(e) = x\}$. Conditional independence can be extended to random variables so that X and Y are conditionally independent given Z if $P(X = x, Y = y|Z = z) = P(X = x|Z = z)P(Y = y|Z = z)$ for $x, y, z \in V$, assuming X, Y, and Z range over the same value space V.

A system is often described in terms of a vector of random variables $X_1, \ldots, X_n$. These have a joint distribution $P(\overline{X} = \overline{x})$. For some inference tasks, the vector of random variables will be divided into two classes: the observable variables $\overline{X}$ and the hidden variables $\overline{Y}$. For example, in classification problems, the random variables consist of a class Y and the observations $X_1, \ldots, X_n$. The objective is to determine $P(Y|\overline{X} = \overline{x})$. Bayes theorem can be used to recompute this as $P(\overline{X} = \overline{x}|Y = y)P(Y = y)/P(\overline{X} = \overline{x})$. If we assume that X_i and X_j, with $i \neq j$, are conditionally independent with respect to Y, then we get

$P(Y = y|\overline{X} = \overline{x}) = \frac{1}{Z}P(Y = y)\prod_{i=1}^{n} P(X_i = x_i|Y = y)$. The naïve Bayes classifier is based on this assumption of conditional independence between the observations. The classification can then be chosen as the y such that $P(Y = y|\overline{X} = \overline{x})$ is maximal.

In some systems, there is a third set of *internal* variables $\overline{W}$ that are neither hidden nor observed and could be parameters to the model. There are four basic inference problems associated with probability:

1. Marginal probabilities: $P(X_i = x) = \sum_{\overline{x}:\overline{x}_i=x} P(\overline{X} = \overline{x})$. The marginal probability isolates the probability distribution for a single random variable (or a subset of the random variables) by marginalizing over the distributions of the other variables.
2. Most Likely Explanation (MLE): $argmax_y Pr(\overline{Y} = \overline{y}|\overline{X} = \overline{x})$. The MLE problem is that of finding the most likely assignment of the hidden variables given a specific observation.
3. Maximum a Posteriori (MAP): $argmax_y \sum_{\overline{w}} Pr(\overline{Y} = \overline{y}|\overline{X} = \overline{x}, \overline{W} = \overline{w})$. The MAP problem is that of finding the most likely assignment of a subset of the hidden variables given a specific observation.
4. Conditional distribution: $P(\overline{Y} = \overline{y}|\overline{X} = \overline{x}) = \frac{\sum_{\overline{w}} P(\overline{Y}=\overline{y},\overline{W}=\overline{w},\overline{X}=\overline{x})}{P(\overline{X}=\overline{x})}$. The conditional probability computes the probability distribution of the hidden variables given the observations while marginalizing out the internal variables.

Techniques for carrying out the above forms of inference are quantitative versions of logical inference, and logic can be used as a knowledge representation framework for probabilistic models. The first step is to build a compact representation of a probabilistic knowledge base involving a system of random variables. For this, joint probability distributions are unduly verbose. For Boolean random variables, where $V = \{0, 1\}$, there are 2^n possible choices for x. *Graphical models* exploit conditional independence to describe the joint probability more compactly [4,20]. A *Bayesian network* [7,26,29] is a directed acyclic graph G consisting of nodes X_i, where each X_i is a random variable. To each node X_i, there is a set of parent variables $\pi(X_i)$ with an edge from each X_j, for X_j in $\pi(X_i)$, to X_i. In a Bayesian network, the joint probability distribution can be written as $P(\overline{X} = \overline{x}) = \prod_i p_i(\overline{x})$, where

$$p_i(\overline{x}) = P(X_i = x_i| \bigwedge_{\{X_j \in \pi(X_i)\}} X_j = x_j)$$

In *undirected* graphical models such as Markov random fields, the random variables are connected by a set of cliques Γ, where each clique C is a subset of $\overline{X}$ and $\bigcup \Gamma = \overline{X}$. To each clique C in Γ, there is a joint distribution F_C over the variables in C. Since each random variable is conditionally independent of other random variables given its neighbors in the graph, the overall joint distribution is computed from the clique-wise joint distributions as $P(\overline{X} = \overline{x}) = \frac{1}{Z}\prod_{\{C\in\Gamma\}} F_C(X_C = x_C)$, where X_C and x_C are $\overline{X}$ and $\overline{x}$ restricted to C. The normalization constant $Z = \sum_{\overline{x}} \prod_{\{C\in\Gamma\}} F_C(X_C = x_C)$.

Factor graphs are bipartite graphs with factor nodes $f_1, \ldots, f_m$ in addition to random variables $X_1, \ldots, X_n$. Let $\overline{X}^j$ represent the subset containing those variables that have edges to f_j in the factor graph. The joint probability distribution is then computed as $P(\overline{X} = \overline{x}) = \frac{1}{Z} \prod_{j=1}^{m} f_j(\overline{X}^j)$, where Z is a normalization constant that ensures that the summation of the joint distribution is 1. The set of factor nodes connected to a random variable X is $N(X)$ and the set of random variables connected to a factor node f is $M(f)$. Both directed and undirected graphical models can be represented as factor graphs. For tree-shaped factor graphs, marginal and conditional probabilities can be computed exactly by means of *variable elimination* and *message passing* algorithms. Approximate algorithms based on Markov Chain Monte Carlo algorithms such as Gibbs sampling and Metropolis-Hastings sampling define random walks through the sample space of assignments for the random variables [25]. MCMC algorithms have several advantages. They are anytime algorithms that deliver approximate answers. It is easy to estimate marginal distributions and expected values of random variables by tabulating the values assigned to the variables in the generated samples.

Potential functions in a graphical model are more conveniently represented as feature functions f_k using a log-linear model as $P(\overline{X} = \overline{x}) = \frac{1}{Z} e^{(\sum_k w_k f_k(\overline{x}))}$. In *Markov logic* [34,8], the random variables X_i are all Boolean. Each feature function corresponds to the truth value of the k'th formula in the knowledge base under the assignment $\overline{x}$. The weight w_i assigned to a formula is a measure of the incremental log-odds of the formula, as shown below. When the weight is maximal, i.e., ∞, Markov logic reduces to ordinary propositional logic. However, even in this case, it is possible to compute probabilities. For example, the problem at the beginning of this section has no probabilities in the input constraints. In the absence of probabilities, all assignments of truth values to the variables are equally probable. Given a knowledge base KB that is a conjunction of maximally weighted formulas, the probability of a formula F can be defined as

$$\frac{|\{M \mid M \models KB \wedge F\}|}{|\{M \mid M \models KB\}|}$$

If the knowledge base is a conjunction of formulas $F_1 \wedge \ldots \wedge F_n$, where each formula F_i has an associated weight w_i, then we define f_i so that $f_i(M) = 1$ if $M \models F_i$, and $f_i(M) = 0$, otherwise. Models have an associated probability in the KB: $P_{KB}(M)$ given by $\frac{1}{Z} e^{\sum_i w_i f_i(M)}$, where Z is a normalization factor. The probability $P_{KB}(F)$ of a formula F is the aggregate over the models of F: $\sum_{\{M|M \models F\}} P_{KB}(M)$.

Example 2. Let KB be over two variables A and B and contain a single formula $A \vee B$ with weight w.

A	B	$P_{KB}(M)$	$\neg B$
$\top$	$\top$	e^w	0
$\top$	$\bot$	e^w	$\top$
$\bot$	$\top$	e^w	0
$\bot$	$\bot$	e^0	$\top$

The normalization factor $Z = \sum_M e^{\sum_i w_i f_i(M)}$, which is $3e^w + 1$. The formula $\neg B$ has probability $\frac{e^w+1}{3e^w+1}$.

At SRI, we have implemented a solver for Markov Logic Networks called the Probabilistic Consistency Engine (PCE). This system is based on the MC-SAT algorithm of Poon and Domingos [31], which is also implemented in their Alchemy system. MC-SAT takes a set K of weighted clauses as input and generates a sequence of models $\langle x^{(0)}, \ldots, x^{(i)}, \ldots \rangle$. Hard clauses are facts (and negations given by the closed-world assumption) and clauses with maximal weight. The first step in the algorithm is to generate an assignment x^0 using the WalkSAT algorithm [35]. Each iterative step generates an assignment x^{i+1} from x^i by choosing a subset K_i of the clauses satisfied by x^i. A subset $\hat{K}_i$ of K_i is constructed by excluding $\kappa \in K_i$ with weight w from $\hat{K}_i$ with probability e^{-w}. The SampleSAT algorithm [42] blends simulated annealing with WalkSAT to generate a random assignment x^{i+1} for the selected constraints $\hat{K}_i$. PCE has been used for applying domain knowledge to the task of information extraction from natural language text![9].

Prior to the implementation of PCE, we were using the weighted MaxSAT and MaxSMT capability of the Yices solver to carry out the MLE inference [28] for finding the most likely assignment x, i.e., $argmax_x P_{KB}(x)$. MaxSampleSAT [19] is another approach for constructing the most likely assignment. Model counting and weighted model counting [12,11] have also been used for computing the probability of a formula in a knowledge base.

We have only covered the propositional case of probabilistic inference in Markov Logic, but the expressiveness can be enhanced in several directions. Sorts and relations over sorts can be added so that $p(a, b, c)$ represents a relation p across a of sort A, b of sort B, and c of sort C. If the sorts are all finite, then each atom $p(a, b, c)$ can be represented by a distinct Boolean variable. Even though relations make the Markov Logic formulas compact since it is possible to write formulas like $\forall(p : Person).\neg(democrat(p) \wedge republican(p))$. However, inference with this kind of grounding can be expensive. There are two ways to mitigate the cost. One way is by lazily grounding the formulas as needed to solve the query. The other way is through *lifted inference* using more refined representations and rules that operate at the symbolic level (in analogy with resolution).

To summarize, there are strong connections between logical inference and probabilistic inference. Exact computation of probabilities in Markov logic networks is a $\#P$-complete problem. However, there are fast methods based on MCMC sampled that can be used to compute reasonable approximations. Probabilistic programming languages like the stochastic lambda calculus [33] and Church [14] use this kind of sampling as an execution mechanism to make inferences on computations defined over distributions. The PRISM model checker [21] also employs probabilistic inference to compute probabilistic/temporal properties of Markov chains and their extensions.

3 Cueology

In the previous section, we briefly covered probabilistic inference in generative models where we are explicitly modelling the causal links within a system of random variables and using these links to identify probable causes from specific observations. The problem with generative models is that there might be a large number of observable and hidden variabes of which only a small subset might be relevant to a given inference task such as making a reasonably accurate prediction. Discriminative models define classifiers that are trained to make such predictions. In this section, we examine the problem of identifying those observable features that can be used to build minimalist discriminators for making predictions.

To motivate this, we return briefly to the idiosyncracies of human (or even, animal) cognition. Architectures for cognition must be designed to deal with a deluge of sensory input and internally stored data. A sentient agent, and particularly its System 1 component, has to quickly extract the signal and the meaning of the signal from this mountain of data. The meaning or the *significate* is typically something that is hidden or implicit, possibly because it is in the past, the future, or the non-observable present. For example, one might want to make predictions that answer questions such as: *Is the oncoming baseball pitch a slider or a fastball? Do the clouds and wind indicate rain this evening? Will the enemy attack by land or sea? Is North Korea about to conduct a nuclear test? How will the stock market react to the latest employment report?*

Similar questions can also be asked about the hidden present: *Is that an 'a' or an 'o'? Is this lesion malignant? Does the bell indicate feeding time? Is the pasta done, the banana over-ripe, or the coffee stale?*

Reading social signals is an extremely important part of human interaction: *Is my interlocutor happy or bored? Are we talking about the same thing? Have we reached agreement? Is the witness lying? Did I detect a trace of sarcasm?*

Extracting meaning from signals can also be a key to deciphering the past: *Did someone eat my porridge? Can we tell who did it? Did the Neanderthals have a spoken language?*

In each case, we have to somehow divine the hidden information on the basis of limited observations even as we are bombarded with other irrelevant stimuli. For example, it is too late for a batter to react to a pitch after observing that it is a slider or a fastball. When a ball is travelling across 60 feet at 90+ miles an hour, it reaches the batter in less than half a second. Good batters exploit other information such as the wind-up, the release point/angle/speed/spin, to assist in the decision to take a swing at the ball. Even the pitcher's facial expression might be relevant. Conversely, we are often subconsciously generating information through facial expressions, tonal modulation, and body language, that can be perceived by others as an indication of our mood or emotional state. Pitchers might also have "tells", that is, mannerisms that reveal the likely nature of the forthcoming pitch. In some cases, we might wish to mask this information by sending out false signals or no signals (as in the poker face of a card player).

The common factor in the above examples is the use of *cues*: signals that are used to perceive information or to generate it. Cues are an important and ubiquitous part of inference. They are used for recognizing friends, storing and retrieving memories, discerning intent, and much else. Cues are significant enough that they can be studied as an independent subject. We propose a very preliminary framework called *cueology* for such a study of the relationship between implicit information and explicit observation. Our senses are bombarded with information, and we can only be conscious of a small fraction of it. Cues provide a way to focus our consciousness on those signals that carry the most salience. The cognitive cycle uses cues to trigger sensitivity to other cues so that we have accurate information to guide our actions. By integrating cues, we are able to recognize complex patterns such as letters, alphabets, words, pictures, and tasks, and can quickly discriminate between different symbols, moods, or situations.

Cues are also a key component of formal human knowledge. We are really seeking cues when we pose questions such as: How can we recognize the species of a given plant or animal? How can we detect a subatomic particle? How do we tell the temperature or measure blood pressure? Is a given substance acidic or alkaline? Is it edible or poisonous? How do we diagnose a disease from symptoms? How does one infer the intended topic of a search from the query string? What is the best way to prove a given conjecture of a certain form?

Experts in a given field are often well-trained at recognizing cues. A stewardess glances at a passenger's book to guess the best language for communication. An art historian can recognize the painter from a painting or instinctively detect a fake, and a literary sleuth can identify the writer from the text. A good programmer can rapidly identify the bugs in a piece of code. This is because learning and experience have trained the expert to distinguish reliable cues, those that have high predictive and discriminative value, from the unreliable ones.

Cues can also be unreliable. One such difficulty arises in switching from steering left-hand drive cars on the right lane to right-hand drive cars on the left lane. A driver typically cues the position of the car within a lane off the closest lane marker, e.g., the left-side lane marker for a left-hand drive car, and this same cue does not work so well when switching to a right-hand side drive. Magicians are adept at exploiting human cue response to draw attention away from the locus of deception. The conjunction fallacy can be seen as a side effect of this kind of 'miscue' since the information given about Linda is more a cue for her turning out to be an active feminist than a bank teller.

Cues have been studied in a large number of areas, including law [41], economics [22], magic[2], neural computing [1], and sports [6], among many others. A search for the words *cue theory* on Wikipedia yields hundreds of relevant hits. We examine cues from the viewpoint of inference. Here, the key question is: what is a cue and when is one cue better than another? If X and Y are Boolean random variables, then X is a *cue* or *sign* for a *significate* Y in context G if $X = \mathbf{true}$ significantly raises the probability of $Y = \mathbf{true}$. One convenient way of measuring the skew introduced by a cue is the odds of $Y = \mathbf{true}$ which is

[2] See `http://www.nytimes.com/2007/08/21/science/21magic.html`.

given by $P(Y = \textbf{true})/(1 - P(Y = \textbf{true})) = P(Y = \textbf{true})/P(Y = \textbf{false})$. Log-odds, i.e., the logarithm $ln(P(Y = \textbf{true}/P(Y = \textbf{false})))$ is even more convenient since $ln(P(Y = \textbf{true}/P(Y = \textbf{false}))) = -ln(P(Y = \textbf{false}/P(Y = \textbf{true})))$. The strength of a cue X for Y is given by the incremental log-odds, i.e.,

$$ln\left(\frac{P(Y = \textbf{true}|X = \textbf{true}, G)}{P(Y = \textbf{false}|X = \textbf{true}, G)}\right) - ln\left(\frac{P(Y = \textbf{true}|G)}{P(Y = \textbf{false}|G)}\right).$$

Measuring the strength of cue as the incremental log-odds is effective for inferring the presence of the significate Y when we have already observed X. Such a cue can be termed a *post-cue*. For example, an effective post-cue might be one that is rarely present even when the significate is present, but when it is present, it raises the probability of the significate in a significant way. In contrast, a *pre-cue* is one that is used to set up an observation for the significate. In this case, we have to choose whether to observe one cue or another. We might for example only be interested in observing cues that raise the probability of the significate above a threshold. This might be the threshold at which we decide to act as if the significate is present. Then the cost of making an incorrect prediction based on a cue is also relevant. Even when we have two cues that each raise the probability of the significate above the threshold, they might differ in the coverage. Type 1 errors (false positives, or loss of precision) occur when the cue is present, but the significate is absent. The probability of Type 1 errors is given by $P(Y = \textbf{false}|X = \textbf{true})$, and the probability of the Type 1 non-error case is given by $P(Y = \textbf{true}|X = \textbf{true})$. Type 2 errors (false negatives, or loss of recall) occur when the cue is absent, but the significate is present. The probability of Type 2 errors is $P(Y = \textbf{true}|X = \textbf{false})$, and the probability of the non-error case is given by $P(Y = false|X = false)$. A cue X measured by incremental log-odds might minimize Type 1 errors at the expense of Type 2 errors. For example, we might have a diagnosis for cancer that strongly indicates cancer, but it does so only in a small number of actual cases of cancer, while a different diagnosis might covers a larger number of actual cases but with a large false positive rate. There is a cost even with correct decisions which might be the cost of observing the cue. If E_1 and C_1 are the costs associated with Type 1 error and non-error decisions, and E_2 and C_2 are the costs associated with the Type 2 error and non-error decisions, then the cue must be chosen to minimize the expected cost.

A cue X can also be used to obtain a better distribution of the significate variable Y. For example, we might be able to discern from the way a baby is crying, that she is either sleepy or hungry, but is more likely to be sleepy than hungry. The cue can then be added to the context in the hope of uncovering some other useful cue in this new context. Typically, the cue X will be used to narrow the distribution of a parameter Z, where we know $P(Y = y|W = w)$. In such cases, the greater discriminating power of the cue is given by the Kullback-Leibler (KL) divergence which is given by $\sum_y P(Y = y|X = x) ln\left(\frac{P(Y=y|X=x)}{P(Y=y)}\right)$, where the probability $P(Y = y|X = x)$ is obtained by marginalizing over the parameter W.

The above discussion emphasizes the analytic aspect of using cues to uncover hidden information, but there is also a synthetic aspect to cues. In the synthetic form, cues are used to implicitly communicate hidden information or to create certain associations. Advertising is a ubiquitous application of synthetic cueology. The distinctive shape of a soft-drink bottle, the deep reddish-orange hue of a laundry detergent, or the swoosh of a sporting goods maker, are cues that create strong associations with specific products, and help discriminate these products from their competition. Synthetic cues are used in language for communicating mental states. They are also helpful in designing choice architectures that subtly bias user behavior [39].

We have outlined a framework called *cueology* that emphasizes the use of signs or cues for extracting the hidden significate. The main thesis of cueology is that hidden information can typically be uncovered by identifying and observing the relevant cues. For natural cognition, we can state a *strong cueing hypothesis* that *the nervous system is a cueing network that integrates sensory and non-sensory (e.g., memory-based) cues.*

Cueology also raises the engineering challenge of building cue-based architectures for fast, parallel inference. One of the challenges for building such an architecture is *feature identification*. What features of a domain are relevant for constructing simple but highly discriminative cues. The other challenge is that of determining from the data, the exact contexts in which a specific cue is effective. Some cues are helpful in isolating situations in which other cues can be effective. A cueing architecture filters the data for certain cues and triggers other cue filters based on the cues that have been detected. The architecture also analyzes models and data, offline or online, to identify useful cueing relations.

In Todd and Gigerenzer's words [40,10], cues offer a *fast and frugal* heuristic for inference and reasoning. They are a way of quickly filtering the signal from the noise. Gigerenzer and his colleagues [10] have mostly examined cues from a cognitive viewpoint, but cues have a clear role to play in more rational forms of inference, and there is a science underlying the use of cueing relations in this kind of inference. In the next section, we argue that cues have a role to play in automated reasoning as well.

4 Discussion

> *Polya showed that even a pure mathematician actually uses these weaker forms of reasoning most of the time. Of course when he publishes a new theorem he will try very hard to invent an argument which uses only the first kind but the reasoning process which led him to the theorem in the first place almost always involves one of the weaker forms...*
>
> E. T. Jaynes, *Probability Theory: The Logic of Science* [15]

Having seen some of the background in probabilistic inference with both generative and discriminative models, and the connections to logical reasoning, we can now examine the argument that automated reasoning can be diversified to incorporate broader forms of inference. While there are connections to human

cognition, the emphasis here is not on modelling or explaining human performance but on developing a rational, high-performance architecture for inference tasks that combine various forms of inference: deductive, inductive, and abductive. Specifically, the dual-process models integrating System 1 and System 2 reasoning can be emulated in artificial systems as well.

Inference tasks need deductive reasoning. For human cognition, this point is quite controversial (see Pinker [30]). However, if we want high performance in terms of both efficiency and accuracy of the results, then deductive reasoning is indispensable. Many inference problems involve a rich set of constraints that include hard (e.g., logical) and soft (e.g., probabilistic) rules as well as observational data. Planning is a typical example, where a plan must satisfy certain logical constraints but it must also cope with uncertainty in the operating environment and the flexibility afforded by the soft rules and objectives. In the future, many complex problems are going to be addressed with cognitive, inference-based solutions, and the relevant inference architectures will have to incorporate a mix of logical, inductive, and abductive inference.

Deduction can itself exploit broader forms of inference. Many deductive inference procedures already exploit heuristics. For example, SAT solvers use activity-based heuristics like VSIDS for prioritizing the search decisions among the unassigned variables. Machine learning techniques are already being used to construct useful interpolants [36]. Saturation-based theorem provers must decide on orderings and strategies. Induction theorem provers guess the induction scheme based on the recursion schemes of the function symbols that appear in the conjecture. Synthesis as a way of instantiating first-order or higher-order quantifiers often exploits inductive reasoning for generalizing from instances. Loosely constrained synthesis tasks, particularly those that involve templates, can be solved by MCMC sampling to identify the parameters of the template. Template-based invariant generation is an example where such probabilistic search can be effective.

Deductive inference itself can exploit cues. By now we have accumulated a large body of data in the form of definitions, theorems, and proofs. We typically do not retain information from failed proof attempts, but this too might be useful. This data can be mined in order to identify the features that can serve as cues for specific inference strategies. It can also be used to pick out the lemmas that might be relevant to the current proof attempt. The rippling heuristic for induction [5] is an example where the simplifications are classified in terms of the relevance for bridging the gap between the induction hypothesis and the induction goal. Other cues might help highlight how universal quantifiers can be processed, e.g., by induction or Skolemization, or if an existential quantifier should be eliminated or explicitly instantiated. The proof data can also be mined for identifying popular inference steps, both individual steps as well as combinations, given the features of the goal.

Human cognitive architectures offer useful lessons for artificial inference. Modern multi-core architectures can be exploited to carry out the analog of System 1 reasoning with a massive degree of parallelism. Such a System 1 could be used to identify promising inference strategies and to extract the relevant background

information. In many cases, System 1 might be able to solve the inference problem. For computational, as opposed to cognitive, processing, even System 2 reasoning can be parallelized. Stanovich, West, and Toplak [16] outline a tripartite architecture which further decomposes the System 2 layer into an algorithmic mind and a reflective mind that questions the goals, beliefs, and assumptions that are employed by the algorithmic mind. Adding a reflective layer to an automated reasoning system remains a difficult and intriguing challenge.

5 Conclusions

A large fraction of computing is about inference, and deductive inference is central to computational and mathematical reasoning with hard constraints. Increasingly, computational problems involve reasoning with uncertainty constrained by rules and probability distributions drawn from the data. Deductive techniques have already proved quite useful in such reasoning tasks, and we need to continue to diversify these techniques to address these newer challenges. We also need to exploit the availability of large volumes of data and the associated analytical techniques for mining this data, to enhance the effectiveness of deductive inference itself. Such a combination of analytical and empirical inference can offer an interface layer for bridging the gap between problems and efficient inference techniques.

From the point of view of purely deductive reasoning, it makes sense to develop architectures that combine fast, approximate, cue-triggered, rule-based reasoners with slower, algorithmic, and reflective ones. The use of iterative abstraction refinement can be seen as combinations of fast reasoning with abstractions, and slow reasoning for refining the level of abstraction and inference when fast reasoning fails.

Cues are used in both natural and artificial cognition to discern and convey information. Inference plays a central role in identifying cues and integrating information derived from multiple cues. We propose cueology as a unified, interdisciplinary study of cues.

References

1. Baldi, P., Itti, L.: Of bits and wows: A Bayesian theory of surprise with applications to attention. Neural Networks 23, 649–666 (2010)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer (2004), Coq home page, `http://coq.inria.fr/`
3. Bevier, W.R., Hunt, W.A., Moore, J.S., Young, W.D.: Special issue on system verification. Journal of Automated Reasoning 5(4), 409–530 (1989)
4. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2006)
5. Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., Smaill, A.: Rippling: A heuristic for guiding inductive proofs. Artif. Intell. 62(2), 185–253 (1993)
6. Cork, A., Justham, L., West, A.: Cricket batting stroke timing of a batsman when facing a bowler and a bowling machine. In: The Engineering of Sport 7, pp. 143–150. Springer (2008)

7. Darwiche, A.: Bayesian networks. Commun. ACM 53(12), 80–90 (2010)
8. Domingos, P., Lowd, D.: Markov Logic: An Interface Layer for Artificial Intelligence. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers (2009)
9. Ghosh, S., Shankar, N., Owre, S.: Machine reading using Markov logic networks for collective probabilistic inference. In: Proceedings of ECML-CoLISD 2011 (2011)
10. Gigerenzer, G., Todd, P.M.: Simple heuristics that make us smart. Oxford University Press, USA (1999)
11. Gogate, V., Domingos, P.: Probabilistic theorem proving. CoRR, abs/1202.3724 (2012)
12. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 633–654. IOS Press (2009)
13. Gonthier, G.: Engineering mathematics: the odd order theorem proof. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, Rome, Italy, January 23-25, pp. 1–2. ACM (2013)
14. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. In: McAllester, D.A., Myllymäki, P. (eds.) Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, UAI 2008, Helsinki, Finland, July 9-12, pp. 220–229. AUAI Press (2008)
15. Jaynes: Probability Theory: The Logic of Science. Cambridge University Press (2003)
16. Toplak, M.E., Stanovich, K.E., West, R.F.: Intelligence and rationality. In: Sternberg, R., Kaufman, S.B. (eds.) Cambridge handbook of intelligence, 3rd edn., pp. 784–826. Cambridge University Press (2012)
17. Kahneman, D.: Thinking, fast and slow. Farrar, Straus and Giroux (2011)
18. Kandel, E.R.: In search of memory: The emergence of a new science of mind. WW Norton & Company (2007)
19. Kautz, H., Selman, B., Jiang, Y.: A general stochastic approach to solving problems with hard and soft constraints. In: The Satisfiability Problem: Theory and Applications, pp. 573–586. AMS (1997)
20. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press (2009)
21. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. International Journal on Software Tools for Technology Transfer (STTT) (2004)
22. Laibson, D.: A cue-theory of consumption. The Quarterly Journal of Economics 116(1) (February 2001)
23. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
24. MacKay, D.J.C.: Information theory, inference, and learning algorithms. Cambridge University Press (2003)
25. Neal, R.M.: Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto (September 1993)
26. Neapolitan, R.E.: Learning Bayesian Networks. Prentice-Hall (2003)
27. Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. IEEE Transactions on Software Engineering 21(2), 107–125 (1995) PVS home page, `http://pvs.csl.sri.com`

28. Park, J.D.: Using weighted MAX-SAT engines to solve MPE. In: Dechter, R., Sutton, R.S. (eds.) Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, Edmonton, Alberta, Canada, July 28-August 1, pp. 682–687. AAAI Press / The MIT Press (2002)
29. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, San Mateo (1988)
30. Pinker, S.: How the mind works. Annals of the New York Academy of Sciences 882(1), 119–127 (1999)
31. Poon, H., Domingos, P.: Sound and efficient inference with probabilistic and deterministic dependencies. In: Proceedings of the Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, Boston, Massachusetts, USA, July 16-20, pp. 458–463. AAAI Press (2006)
32. Ramsey, F.P.: Truth and probabiity. In: Mellor, D.H. (ed.) Philosophical Papers, pp. 52–109. Cambridge University Press (1990)
33. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: Conf. Record of 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2002, pp. 154–165. ACM Press, New York (2002)
34. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning 62(1-2), 107–136 (2006)
35. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: AAAI-92: Proceedings 10th National Conference on AI. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (January 1995)
36. Sharma, R., Nori, A.V., Aiken, A.: Interpolants as classifiers. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 71–87. Springer, Heidelberg (2012)
37. Silver, N.: The Signal and the Noise: Why So Many Predictions Fail-but Some Don't. Penguin Press (2012)
38. Simon, H.A.: What is an "explanation" of behavior? Psychological Science 3(3), 150–161 (1992)
39. Thaler, R.H., Sunstein, C.R.: Nudge: Improving decisions about health, wealth, and happiness. Yale University Press (2008)
40. P.M. Todd, G. Gigerenzer. Précis of simple heuristics that make us smart. *Behavioral and brain sciences*, 23(05):727–741, 2000.
41. Ulmer, S.S., Hintze, W., Kirklosky, L.: The decision to grant or deny certiorari: Further consideration of cue theory. Law & Society Review 6(4), 637–644 (1972)
42. Wei, W., Erenrich, J., Selman, B.: Towards efficient sampling: Exploiting random walk strategies. In: McGuinness, D.L., Ferguson, G. (eds.) Proceedings of the Nineteenth National Conference on Artificial Intelligence, pp. 670–676. AAAI Press / The MIT Press (2004)

Foundational Proof Certificates in First-Order Logic

Zakaria Chihani, Dale Miller, and Fabien Renaud

INRIA and LIX, Ecole Polytechnique, Palaiseau, France

Abstract. It is the exception that provers share and trust each others proofs. One reason for this is that different provers structure their proof evidence in remarkably different ways, including, for example, proof scripts, resolution refutations, tableaux, Herbrand expansions, natural deductions, etc. In this paper, we propose an approach to *foundational proof certificates* as a means of flexibly presenting proof evidence so that a relatively simple and universal proof checker can check that a certificate does, indeed, elaborate to a formal proof. While we shall limit ourselves to first-order logic in this paper, we shall not limit ourselves in many other ways. Our framework for defining and checking proof certificates will work with classical and intuitionistic logics and with proof structures as diverse as resolution refutations, matings, and natural deduction.

1 Introduction

Consider a world where the multitude of computational logic systems—theorem provers, model checkers, type checkers, static analyzers, etc.—can trust each other's proofs. Such a world can be constructed if computational logic systems can output their proof evidence in documents with a clear semantics that can be validated by a trusted checker. By the term *proof certificate* we shall mean documents that contain the evidence of proof generated by a theorem prover. In this paper, we propose a framework for defining the semantics of a wide range of proof evidence using proof-theoretic concepts. As a result, we refer to this approach to defining certificates as "foundational" since it is based not on the technology used to construct a specific theorem prover but rather on basic insights into the nature of proofs provided by the modern literature of proof theory.

The key concept that we take from proof theory is that of *focused proof systems* [1,15,16]. Such proof systems exist for classical, intuitionistic, and linear logics and they are composed of alternating *asynchronous* and *synchronous* phases. These two phases allow for a natural interaction to be set up between a process that is attempting to build a proof (the checker) and the information contained in a certificate. During the asynchronous phase of proof construction, the checker proceeds without reference to the actual certificate since this phase consists of invertible inference rules. During the synchronous phase, information from the certificate can be extracted to guide the construction of the focused proof. The definition of how to check a proof certificate essentially boils down to defining the details of this interaction.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 162–177, 2013.

The main structure for our framework contains the following components.

- The *kernel* of our checker is a logic program specification of the *focusing framework LKU* proof system [16]. Since this implementation of *LKU* is high-level and direct, we can have a high degree of confidence that the program does, in fact, capture the *LKU* proof system.
- By restricting various structural rules, *LKU* can be made into a focused proof system for classical logic, for intuitionistic logic, and for multiplicative-additive linear logic. The specifications of these restrictions are contained in separate small *logic definition* documents.
- The kernel implementation of *LKU* actually adds another premise to every inference rule: in particular, the asynchronous rules get a premise involving a *clerk predicate* that simply manages some bookkeeping computations while the synchronous rules get a premise involving an *expert predicate* that extracts information from the certificate to provide to the inference rule. A *proof certificate definition* is a document that defines these two kinds of predicates as well as a translation function from theorems of the considered system to equiprovable LKU formulas.
- A *proof certificate* is a document consisting of the structured object containing the proof evidence supporting theoremhood for a particular formula.

To illustrate this architecture, we present a number of different proof certificates. For example, a certificate for resolution refutations can be taken as a list of clauses (including those arising from the original theorem and those added during resolutions) and a list of triples that describes which two clauses resolve to yield a third clause. Such an object should be easy to produce for any theorem prover that uses binary resolution (with implicit factoring). By then adding to the kernel the logic definition for classical logic (given in [16]) and the definitions of the clerk and expert predicates (given in Section 4.3), resolution refutations can be checked. The exact same kernel (this time restricted to intuitionistic logic) can be used to check natural deduction proofs (*i.e.*, simply and dependently typed λ-terms): all that needs to be changed is the definition of the clerk and expert predicate definitions.

Before presenting specific examples of proof certificate definitions for first-order classical logic in Section 4, we describe focused proof systems in the next section and, in Section 3, we describe how we have augmented and implemented that proof system within logic programming. The current implementation of our proof checking system is available at `https://team.inria.fr/parsifal/proofcert/`.

2 Proof Theory Architecture

The sequent calculus of Gentzen [10] (which we assume is familiar to the reader) is an appealing setting for starting a discussion of proof certificates. First of all, sequent calculus is well studied and applicable to a wide range of logics. The introduction rules, structural rules (weakening and contraction), and the identity

rules (initial and cut) provide a convincing collection of "atoms" of inference. Additionally, cut-elimination theorems are deep results about sequent calculus proof systems that not only prove them to be consistent but also offers *cut-free proofs* as a normal form for proof. Girard's invention of linear logic [11] provides additional extensions to our understanding of the sequent calculus, including such notions as *additive* and *multiplicative* connectives, *exponentials*, and *polarities*. Finally, this foundation of Gentzen and Girard lifts naturally and modularly to higher-order logic and to inductive and coinductive fixed points (such as Baelde's μMALL [4]). In this paper, we shall concentrate on first-order (and propositional) logic: we leave the development of proof certificates for higher-order quantification and fixed points for later work.

The sequent calculus has a serious downside, however: sequent proofs are far too unstructured to directly support almost any application to computer science. What one needs is a flexible way to organize the "atoms of inference" into much larger and rigid "molecules of inference." The hope would be, of course, that these larger inference rules can be structured to mimic the notion of proof found in computational logic systems. For example, early work on the proof-theoretic foundations of logic programming [19] showed how sequent calculus proofs representing logic programming executions could be built using two alternating phases: the *backchaining* phase is a focused application of left-rules and the *goal-reduction* phase is a collection of right-rules. Andreoli [1] generalized that earlier work by introducing *focused proofs* for linear logic in which such phases were directly captured and generalized. Subsequently, Liang & Miller presented the *LKF* and *LJF* focused proof systems for classical and intuitionistic logics [15] and later the *LKU* proof system [16] that unified *LKF* and *LJF*. While our current approach to foundational proof certificates is based on *LKU* (allowing the checking of proofs in classical as well as intuitionistic logic), we shall illustrate our approach by considering the simpler *LKF* subsystem of *LKU*. Before presenting *LKF* in detail, we consider the following few elements of its design.

Additive vs multiplicative rules. We shall use t, f, $\wedge$, $\vee$, $\forall$, and $\exists$ as the logical connectives of first-order classical logic and sequents will be one-sided. As is familiar to those working with the sequent calculus, there is a choice to make between using the additive and multiplicative versions of the binary connective $\wedge$ and $\vee$ (and their units t and f, respectively): the most striking difference between these two versions is illustrated with $\vee$:

$$\text{Additive: } \frac{\vdash \Theta, B_i}{\vdash \Theta, B_1 \vee B_2}\ i \in \{1,2\} \qquad \text{Multiplicative: } \frac{\vdash \Theta, B_1, B_2}{\vdash \Theta, B_1 \vee B_2}$$

These two inference rules are inter-admissible in the presence of contraction and weakening. For this reason, one usually selects one of these inference rules and discards the other one. In isolation, however, these inference rules are strikingly different: the multiplicative version is invertible while the additive version reveals that one disjunct is not needed at this point of the proof. The *LKF* proof system will contain the additive and multiplicative versions of disjunction, conjunction, truth, and false: their presence will improve our flexibility for describing proofs.

$$\frac{}{\vdash \Theta \Uparrow t^-, \Gamma} \qquad \frac{\vdash \Theta \Uparrow A, \Gamma \quad \vdash \Theta \Uparrow B, \Gamma}{\vdash \Theta \Uparrow A \wedge^- B, \Gamma} \qquad \frac{\vdash \Theta \Uparrow \Gamma}{\vdash \Theta \Uparrow f^-, \Gamma} \qquad \frac{\vdash \Theta \Uparrow A, B, \Gamma}{\vdash \Theta \Uparrow A \vee^- B, \Gamma} \qquad \frac{\vdash \Theta \Downarrow [t/x]B}{\vdash \Theta \Downarrow \exists x.B}$$

$$\frac{}{\vdash \Theta \Downarrow t^+} \qquad \frac{\vdash \Theta \Downarrow B_1 \quad \vdash \Theta \Downarrow B_2}{\vdash \Theta \Downarrow B_1 \wedge^+ B_2} \qquad \frac{\vdash \Theta \Downarrow B_i \quad i \in \{1, 2\}}{\vdash \Theta \Downarrow B_1 \vee^+ B_2}$$

$$\frac{\vdash \Theta \Uparrow [y/x]B, \Gamma \quad y \text{ not free in } \Theta, \Gamma, B}{\vdash \Theta \Uparrow \forall x.B, \Gamma} \qquad \frac{}{\vdash \neg P_a, \Theta \Downarrow P_a}\ \mathit{init} \qquad \frac{\vdash \Theta \Uparrow B \quad \vdash \Theta \Uparrow \neg B}{\vdash \Theta \Uparrow \cdot}\ \mathit{cut}$$

$$\frac{\vdash \Theta, C \Uparrow \Gamma}{\vdash \Theta \Uparrow C, \Gamma}\ \mathit{store} \qquad \frac{\vdash \Theta \Uparrow N}{\vdash \Theta \Downarrow N}\ \mathit{release} \qquad \frac{\vdash P, \Theta \Downarrow P}{\vdash P, \Theta \Uparrow \cdot}\ \mathit{decide}$$

Here, P is a positive formula; N a negative formula; P_a a positive literal; C a positive formula or negative literal; and $\neg B$ is the negation normal form of the negation of B.

Fig. 1. *LKF*: a focused proof systems for classical logic

Polarized connectives. We *polarize* the propositional connectives as follows: those inference rules that are invertible introduce the negative version of the connective while those inference rules that are not necessarily invertible introduce the positive version of the connective. Thus the additive rule above for the disjunction introduces $\vee^+$ while the multiplicative rule introduces $\vee^-$. The universal quantifier is obviously polarized negatively while the existential quantifier is polarized positively. Literals must also be polarized: these can be polarized in an arbitrary fashion as long as complementing a literal also flips its polarity. We say that a non-literal formula is positive or negative depending only on the polarity of its top-level connective.

Phases organize groups of inference rules. The inference rules for *LKF* are given in Figure 1. Notice that these inference rules involve sequents of the form $\vdash \Theta \Uparrow \Gamma$ and $\vdash \Theta \Downarrow B$ where Θ is a multiset of formulas, Γ is a list of formulas, and B is a formula. Such sequents can be approximated as the one-sided sequents $\vdash \Theta, \Gamma$ and $\vdash \Theta, B$, respectively. Furthermore, introduction rules are applied to either the first element of the list Γ in the $\Uparrow$ sequent or the formula B in the $\Downarrow$ sequent. This occurrence of the formula B is called the *focus* of that sequent. Proofs in *LKF* are built using two kinds of alternating *phases*. The *asynchronous* phase is composed of invertible inference rules and only involves $\Uparrow$-sequents in the conclusion and premise. The other kind of phase is the *synchronous* phase: here, rule applications of such inference rules often require choices. In particular, the introduction rule for the disjunction requires selecting either the left or right disjunct and the introduction rule for the existential quantifier requires selecting a term for instantiating the quantifier. The initial rule can terminate a synchronous phase and the cut rule can restart an asynchronous phase. Finally, there are three structural rules in *LKF*. The *store* rule recognizes that the first formula to the right of the $\Uparrow$ is either a negative atom or a positive formula: such a formula does not have an invertible inference rule and, hence, its treatment is delayed by storing it on the left. The *release* rule is used when the formula under focus (*i.e.*, the formula to the right of the $\Downarrow$) is no longer positive: at such a moment,

the phase changes to the asynchronous phase. Finally, the *decide* rule is used at the end of the asynchronous phase to start a synchronous phase by selecting a previously stored positive formula as the new *focus*.

Impact of the polarity assignment. Let B be a first-order formula and let $\hat{B}$ result from B by placing either $+$ or $-$ on occurrences of t, f, $\wedge$, and $\vee$ (there are exponentially many such placements). It is proved in [15] that B is a classical theorem if and only if $\vdash \cdot \Uparrow \hat{B}$ has an *LKF* proof. Thus the different polarizations do not change *provability* but can radically change the structure of proofs. A simple induction on the structure of an *LKF* proof of $\vdash \cdot \Uparrow B$ (for some polarized formula B) reveals that every formula that occurs to the left of $\Uparrow$ or $\Downarrow$ in one of its sequents is either a negative literal or a positive formula. Also, it is immediate that the only occurrence of a contraction rule is within the decide rule: thus, only the positive formulas are contracted. Since there is flexibility in how formulas are polarized, the choice of polarization can, at times, lead to greatly reduced opportunities for contraction. When one is able to eliminate or constrain contractions, naive proof search can sometimes become a decision procedure.

3 Software Architecture

Of the many qualities that we might want for a proof checker—universality, flexibility, efficiency, *etc.*—the one quality on which no compromise is possible is that of *soundness*. If we cannot prove or forcefully argue for the soundness of our checkers, then this project is without *reason d'être*.

3.1 Programming Language Support

An early framework for building sound proof checkers was the "Logic of Computable Functions" (LCF) system of Gordon, Milner, and Wadsworth [12]. In that framework, the ML programming language was created in order to support the task of building and checking proofs in LCF with a computing facility that provided strong typing and the abstractions associated to higher-order programming and abstract datatypes. Given the design of ML, it was possible to declare a type of theorems, say, `thm`, and to admit certain functions that are allowed to build elements of type `thm` (these encode axioms and inference rules). These latter functions could then be bundled into an abstract datatype and the programming language would enforce that the only items that eventually were shown to have type `thm` were those that ultimately were constructed from the axioms and inference rules encoded into the theorem abstract datatype. Of course, trusting that a checker written in this approach to LCF meant also trusting that (1) ML had the *type preservation property* and (2) the language implementation was, in fact, correct for the intended semantics (*i.e.*, that the addition function translated to the intended addition function, etc.).

This ML/LCF approach to proof checking is based on the most simple notion of proof (variously named after Hilbert or Frege) as a linear sequence of formulas arising from axioms and applications of inference rules.

$$\forall\Theta\forall\Gamma.\ \texttt{async}(\Theta, [t^-|\Gamma]).$$
$$\forall\Theta\forall\Gamma\forall A\forall B.\ \texttt{async}(\Theta, [(A \wedge^- B)|\Gamma])\ \texttt{:-}\ \texttt{async}(\Theta, [A|\Gamma]),\ \texttt{async}(\Theta, [B|\Gamma]).$$
$$\forall\Theta\forall\Gamma\forall A\forall B.\ \texttt{sync}(\Theta, A \vee^+ B)\ \texttt{:-}\ \texttt{sync}(\Theta, A);\ \texttt{sync}(\Theta, B).$$
$$\forall\Theta\forall\Gamma\forall P.\ \texttt{async}(\Theta, [])\ \texttt{:-}\ \texttt{memb}(P, \Theta),\ \texttt{pos}(P),\ \texttt{sync}(\Theta, P).$$
$$\forall\Theta\forall B\forall C.\ \texttt{async}(\Theta, [])\ \texttt{:-}\ \texttt{negate}(B, C),\ \texttt{async}(\Theta, B),\ \texttt{async}(\Theta, C).$$

Fig. 2. Five logic programming clauses specifying *LKF* inference rules

The material in Section 2 illustrates that there can be a great deal more to the structure of proof than is available in such linear proof structures. We are fortunate that in order to take advantage of that rich structure, we do not need to invent a meta-language (in the sense that ML was invented to support LCF): an appropriate meta-language already exists in the λProlog programming language [18]. In contrast to the functional programming language ML, λProlog is a logic programming language. Like ML, λProlog is also strongly typed and has both higher-order programming and abstract datatypes. λProlog has a number of features that should make it a superior proof checker when compared with ML. In particular, λProlog's operational semantics is based on search and backtracking: this is in contrast to the notion of exception handling that is part of the non-functional side of ML. Furthermore, λProlog comes with much more of logic built into the language: in particular, it contains a logically sound notion of unification and substitution for expressions involving bindings (these latter features of λProlog are not generally provided by Prolog).

Although we shall not assume that the reader is familiar with λProlog, familiarity with the general notions of logic programming is particularly relevant to proof checking. Notice that it is nearly immediate to write a logic program that captures the *LKF* proof system in Figure 1. First select two binary predicates, say $\texttt{async}(\cdot,\cdot)$ and $\texttt{sync}(\cdot,\cdot)$, denoting the $\Uparrow$ and $\Downarrow$ judgments. Second write one Horn clause for each inference rule: here the conclusion and the premises of a rule correspond to the head and the body of such a clause. (The declarative treatment of the inference rules involving the quantifiers is provided directly by λProlog.) Of the fourteen Horn clauses that correspond to the fourteen inference rules in Figure 1, five are illustrated in Figure 2: these clauses correspond to the introduction rules for t^-, $\wedge^-$, and $\vee^+$ as well as the decide and cut rules. Some additional predicates have been introduced to specify membership in a multiset, the negation of a formula, and determining if a given formula is positive or not.

The full program can easily be seen to be sound in the sense that the sequent $\vdash \cdot \Uparrow B$ has an *LKF* proof if the atom $\texttt{async}([], B)$ has a proof using this logic program. Using standard depth-first search strategies would result, however, in surprisingly few proofs of the atom $\texttt{async}([], B)$: the clauses specifying the cut rule and the decide rule would immediately result in looping computations. We present this logic program not to suggest that it is appropriate for proving theorems but to show how to modify it to make it into a flexible proof checker.

$$\frac{t_e(\Xi)}{\Xi \vdash \Theta \Downarrow t^+} \qquad \frac{\Xi_1 \vdash \Theta \Downarrow B_1 \quad \Xi_2 \vdash \Theta \Downarrow B_2 \quad \wedge_e(\Xi, \Xi_1, \Xi_2)}{\Xi \vdash \Theta \Downarrow B_1 \wedge^+ B_2}$$

$$\frac{\Xi' \vdash \Theta \Downarrow B_i \quad i \in \{1,2\} \quad \vee_e(\Xi, \Xi', i)}{\Xi \vdash \Theta \Downarrow B_1 \vee^+ B_2} \qquad \frac{\Xi' \vdash \Theta \Downarrow [t/x]B \quad \exists_e(\Xi, \Xi', t)}{\Xi \vdash \Theta \Downarrow \exists x.B}$$

$$\frac{\Xi_1 \vdash \Theta \Uparrow B \quad \Xi_2 \vdash \Theta \Uparrow \neg B \quad cut_e(\Xi, \Theta, \Xi_1, \Xi_2, B)}{\Xi \vdash \Theta \Uparrow \cdot}\ cut$$

$$\frac{\Xi' \vdash \Theta \Uparrow N \quad release_e(\Xi, \Xi')}{\Xi \vdash \Theta \Downarrow N}\ release \qquad \frac{init_e(\Xi, \Theta, l) \quad \langle l, \neg P_a \rangle \in \Theta}{\Xi \vdash \Theta \Downarrow P_a}\ init$$

$$\frac{\Xi' \vdash \Theta \Downarrow P \quad decide_e(\Xi, \Theta, \Xi', l) \quad \langle l, P \rangle \in \Theta \quad \text{positive}(P)}{\Xi \vdash \Theta \Uparrow \cdot}\ decide$$

$$\frac{\Xi' \vdash \Theta \Uparrow \Gamma \quad f_c(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow f^-, \Gamma} \qquad \frac{\Xi_1 \vdash \Theta \Uparrow A, \Gamma \quad \Xi_2 \vdash \Theta \Uparrow B, \Gamma \quad \wedge_c(\Xi, \Xi_1, \Xi_2)}{\Xi \vdash \Theta \Uparrow A \wedge^- B, \Gamma}$$

$$\frac{\Xi' \vdash \Theta \Uparrow A, B, \Gamma \quad \vee_c(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow A \vee^- B, \Gamma} \qquad \frac{\Xi' \vdash \Theta \Uparrow [y/x]B, \Gamma \quad \forall_c(\Xi, \Xi') \quad y \text{ not free in } \Xi, \Theta, \Gamma, B}{\Xi \vdash \Theta \Uparrow \forall x.B, \Gamma}$$

$$\frac{}{\Xi \vdash \Theta \Uparrow t^-, \Gamma} \qquad \frac{\Xi' \vdash \Theta, \langle l, C \rangle \Uparrow \Gamma \quad store_c(\Xi, C, \Xi', l)}{\Xi \vdash \Theta \Uparrow C, \Gamma}\ store$$

Fig. 3. The augmented *LKF* proof system LKF^a

3.2 Clerks and Experts

Consider being in possession of a proof certificate of a theorem and being asked to build an *LKF* proof of that theorem. The construction of the asynchronous phase is independent of any proof evidence you have (hence the name "asynchronous" for this phase). At the end of the asynchronous phase, the construction of the *LKF* proof can proceed with either the cut rule or the decide rule: in both cases, genuine information (a cut formula or a focus formula) must be communicated to the checker. Furthermore, the synchronous phase needs to determine which disjunct to discard in the $\vee^+$ rule and which term to use in the $\exists$ rule. To capture this sense of information flowing between a checker and a certificate, we present in Figure 3 an augmented version of *LKF*, called LKF^a. The augmentations to the *LKF* inference rules is done in three simple steps: (*i*) a proof certificate term, denoted by the syntactic variable Ξ is added to every sequent; (*ii*) every inference rule of *LKF* is given an additional premise using either an *expert predicate* or a *clerk predicate*; and (*iii*) the multiset of formulas to the left of the arrows $\Uparrow$ and $\Downarrow$ are extended to be a multiset of pairs of an *index* and a formula. Thus, the *LKF* proof system can be recovered from LKF^a by removing all occurrences of the syntactic variable Ξ and by removing all premises with a subscripted e or c as well as replacing all occurrences of tuples such as $\langle l, B \rangle$ with just B.

The expert predicates are used to intermediate between the needs for information of the cut rule and the synchronous phase and the information that is present in a proof certificate. All of them examine the certificate Ξ and returns the information needed to continue with as many certificates as there are premises in the rules. For example, the disjunction expert returns either 1 or 2

depending on which disjunct this introduction rule should select. The intension of the existential quantifier expert is that it returns a term t that is to be used in this introduction rule. Notice that the conjunction expert does nothing more than determine the proof certificates to be used in its two premises. The expert for the t^+ determines whether or not it should allow the proof checking process to end with this inference rule. The cut expert examines both the proof certificate and the context Θ and extracts the necessary cut formula for that inference rule. Notice that if this predicate is defined to always fail (*i.e.*, it is the empty relation), then checking this certificate will involve only cut-free *LKF* proofs. Finally, the decide expert gives the positive formula with which to start the new asynchronous phase.

The introduction rules of the asynchronous phase are given an additional premise that involves a clerk predicate: these new premises do not extract any information from the certificate but rather they take care of bookkeeping calculations involving the progress of the asynchronous phase. For example, the $\wedge_c(\Xi, \Xi_1, \Xi_2)$ judgment can be used to record in Ξ_1 the fact that proof checking is on the left branch of this conjunction as opposed to the right branch.

One of the strengths of our approach to proof certificates is that experts can be non-deterministic since this allows a trade-off between the size of a certificate and proof-reconstruction time. For example, let Ξ be a particular certificate and consider using it to introduce an existential quantifier. This introduction rule queries the expert $\exists_e(\Xi, \Xi', t)$. If the Ξ certificate explicitly contains the term t, the expert can extract it for use in this inference rules. If the certificate does not contain this term then the judgment $\exists_e(\Xi, \Xi', t)$ could succeed for every term t (and for some Ξ'). In this case, the expert provides no information as to which substitution term to use and, therefore, the certificate can be smaller since it does not need to contain the (potentially large) term t. On the other hand, the checker will need to reconstruct an appropriate such term during the checking process (using, for example, the underlying logic programming mechanism of unification). When experts are queried during the synchronous phase, their answers may be specific, partial, or completely unconstrained.

The three remaining rules (store, init, decide) of LKF^a reveal the structure of the collection of formulas we have been designating with the syntactic variable Θ. In our presentation of the *LKF* proof system, this structure has been taken to be a multiset of formulas. In our augmented proof system, we shall take this sequent context to be a multiset of pairs $\langle I, C\rangle$ where C is a formula and I is an index. When we need to refer to a specific occurrence of a formula in Θ (in, say, the decide rule), an index is used for this purpose. It is the clerk predicate associated to the store inference rule that is responsible for computing the index of the formula when it is moved from the right to the left of the $\Uparrow$. When the expert predicate in the decide rule describes the formula on which to focus, it does so by returning that formula's index. Finally, the initial expert determines which stored negative literal should be the complement of the focused literal. In the augmented form of both the decide and initial rules, additional premises have been added to check that the indexes returned by the expert predicates

are, indeed, indexes for the correct kind of formula: in this way, a badly defined expert cannot lead to the construction of an illegal *LKF* proof.

The structure of the indexing scheme is left open for the certificate definition to describe. As we shall illustrate later, indexes can be based on, for example, de Bruijn numbers, path addresses within a formula, or formulas themselves. It is possible for a formula to occur twice in the context Θ with two different indexes. We shall generally assume, however, that the indexes *functionally determine* formulas: if $\langle l, C_1 \rangle \in \Theta$ and $\langle l, C_2 \rangle \in \Theta$ then C_1 and C_2 are equal.

Assume that we have a logic programming system that provides a sound implementation of Horn clauses (for example, unification contains the occurs-check). A proof of $\Xi \vdash \cdot \Uparrow B$ within a logic programming implementation of LKF^a (along with the programs defining the experts and clerks) immediately yields an *LKF* proof of $\vdash \cdot \Uparrow B$. This follows easily since the logic programming proof of this goal can be mapped to an *LKF* proof directly: the only subtlety being that the mapping from indexes to formulas must be functional so that the indexes returned by the decide and initial rules are given a unique interpretation in the *LKF* proof. Notice that no such *LKF* proof is actually constructed: rather, it is performed. Notice also that this soundness guarantee holds with no restrictions placed on the implementation of the clerk and expert predicates.

3.3 Defining a Proof Certificate Definition

In order to define a proof certificate for a particular format, we first need to translate theorems into LKU formulas. This operation stays outside the kernel and its correctness has to be proved. Furthermore we need to define the specific items that are used to augment *LKF*. In particular, the constructors for proof certificate terms and for indexes must be provided: this is done in λProlog by declaring constructors of the types `cert` and `index`. In addition, the definition must supply the logic program defining the clerk predicates and the expert predicates. Writing no specification for a given predicate defines that predicate to hold for no list of arguments. Figures 4, 5, and 6 are examples of such proof certificate definitions.

4 Some Certificate Definitions for Classical Logic

We now present some proof certificate definitions for classical logic: the first two deal with propositional logic while the third additionally treats first-order quantification. The first step is to define a translation function from classical formulas to LKF formulas. In this case, this boils down to choosing a polarization of the logical connectives and atomic formulas. Our first two examples of proof certificates are based on assigning negative polarizations to all atoms and to all connectives: *i.e.*, we only use $\wedge^-$, $\vee^-$, t^-, and f^-. A useful measurement of an *LKF* proof is its *decide depth*, *i.e.*, the maximum number of instances of the decide rule along any path from the proof's root to one of its leaves.

$$\texttt{cnf} : \texttt{cert} \qquad\qquad \texttt{idx} : \texttt{form} \to \texttt{index}$$

$$\forall C.\ \mathit{store}_c(\texttt{cnf}, C, \texttt{cnf}, \texttt{idx}(C)). \qquad \wedge_c(\texttt{cnf}, \texttt{cnf}, \texttt{cnf}).$$

$$\forall \Theta \forall l.\ \mathit{init}_e(\texttt{cnf}, \Theta, l). \qquad \vee_c(\texttt{cnf}, \texttt{cnf}).$$

$$\forall \Theta \forall l.\ \mathit{decide}_e(\texttt{cnf}, \Theta, \texttt{cnf}, l). \qquad f_c(\texttt{cnf}, \texttt{cnf}).$$

$$\mathit{release}_e(\texttt{cnf}, \texttt{cnf}).$$

Fig. 4. A checker based on a simple decision procedure

4.1 A Decision Procedure

There is a simple decision procedure for checking whether or not a classical propositional formula is a tautology and we can design a proof certificate definition that implements such a decision procedure. This example illustrates an extreme trade-off between certificate size (here, constant-size) and proof reconstruction time (exponential time). In particular, notice that there is an *LKF* proof of a propositional formula if and only if that proof has decide depth 1 (possibly 0 if the formula contains no literals). The structure of an *LKF* proof of a tautology first builds the asynchronous phase, which ends with several premises all of the form $\vdash \mathcal{L} \Uparrow \cdot$ for some multiset of literals $\mathcal{L}$. Such a sequent is provable if and only if $\mathcal{L}$ has complementary literals: in that case, the *LKF* proof is composed of a decide rule (selecting a positive literal) and initial (matching that atom with a negative literal).

This decision procedure can be specified as the proof certificate definition in Figure 4. The single constant `cnf` is used for the certificate and formulas are used to denote indexes (thereby trivializing the notion of indexes) so we need a constructor to coerce formulas into indexes. Figure 4 also contains the specifications of the clerk and expert predicates. Notice that the initial expert does not behave expertly: it relates the `cnf` certificate to all indexes l and all contexts Θ. Our definition of this predicate here can be unconstrained since the index that it returns is not trusted: that is, the initial rule in LKF^a will check that l is the index of the complement of the focus formula. In the usual logic programming sense, the *check* in the premise is all that is necessary to *select* the correct index. A similar statement holds for the decide expert predicate definition.

4.2 Matings

Let B be a classical propositional formula in negation normal form. Andrews defined a *mating* $\mathcal{M}$ for B as a set of complementary pairs of literal occurrences in B [2]. A mating denotes a proof if every *vertical path* in B (read: clause in the conjunctive normal form of B) contains a pair of literal occurrences given by set $\mathcal{M}$. A certificate definition for proof matings is given in Figure 5. Indexes are, in fact, paths in a formula since they form a series of instructions to move left or right through the binary connectives or to stop (presumably at a literal). There are two constructors for the `cert` type: `aphase` is applied to a list of indexes and `sphase` is applied to a single index. These two constructors are used to mimic

$$\begin{array}{rl}
\texttt{root} : \texttt{index} & \texttt{left},\ \texttt{right} : \texttt{index} \to \texttt{index} \\
\texttt{aphase} : \texttt{list index} \to \texttt{cert} & \texttt{sphase} : \texttt{index} \to \texttt{cert} \\
\forall I \forall Is.\ \vee_c(\texttt{aphase}([I|Is]), \texttt{aphase}([\texttt{left}(I), \texttt{right}(I)|Is])). \\
\forall I \forall Is.\ \wedge_c(\texttt{aphase}([I|Is]), \texttt{aphase}([\texttt{left}(I)|Is]), \texttt{aphase}([\texttt{right}(I)|Is])). \\
\forall I \forall Is.\ f_c(\texttt{aphase}([I|Is]), \texttt{aphase}(Is)). \\
\forall C \forall I \forall Is.\ store_c(\texttt{aphase}([I|Is]), C, \texttt{aphase}(Is), I). \\
\forall I.\ release_e(\texttt{sphase}(I), \texttt{aphase}([I])). \\
\forall \Theta \forall l.\ decide_e(\texttt{aphase}([]), \Theta, \texttt{sphase}(l), l) \\
\forall \Theta \forall k \forall l.\ init_e(\texttt{sphase}(k), \Theta, l)\ \texttt{:-}\ \ \langle k, l \rangle \in \mathcal{M}.
\end{array}$$

Fig. 5. Mating certificate definition

(using paths) the *LKF* asynchronous and synchronous sequents (using formulas). The initial expert will only select index l if it is $\mathcal{M}$-mated to the focused formula (with path address k). Here, we have assumed that $\mathcal{M}$ contains ordered pairs of occurrences in which the first occurrence names a positive literal and the second occurrence names a negative literal. Thus, in order to determine if $\mathcal{M}$ is a proof mating for the formula B, set $\hat{B}$ to be the polarization of B using only negative connectives and check if the goal formula $\texttt{async}([\texttt{root}], \hat{B})$ succeeds from the logic program composed of the augmented *LKF* system, the clerk and expert predicate definitions above, and an encoding of the $\langle k, l \rangle \in \mathcal{M}$ predicate.

4.3 Resolution Refutations

A (resolution) clause is a closed formula that is the universal closure of a disjunction of literals (the empty disjunction is false). When we polarize, we use the negative versions of these connectives and we assign negative polarity to atomic formulas. We assume that a certificate for resolution contains the following items: a list of all clauses $C_1, \ldots, C_p$ $(p \geq 0)$; the number $n \geq 0$ which selects the last clause that is part of the original problem (*i.e.*, this certificate is claiming that $\neg C_1 \vee \cdots \vee \neg C_n$ is provable and that $C_{n+1} ... C_p$ are intermediate clauses used to derive the empty one); and a list of triples $\langle i, j, k \rangle$ where each such triple claims that C_k is a binary resolution (with factoring) of C_i and C_j. If the implementer of a resolution prover wished to output refutations, this kind of document should be easy to accommodate.

Checking this structure is done in two steps. First, we check that a particular binary resolution is sound and then we check that the list of resolvents leads to an empty clause. It is a simple matter to prove the following: if clauses C_1 and C_2 yield resolvent C_0 as a binary resolvent (allowing also factoring), then the focused sequent $\vdash \neg C_1, \neg C_2 \Uparrow C_0$ has a proof of decide depth 3 or less. We can also restrict such a proof so that along any path from the root sequent to its leaves, the same clause is not decided on more than once. The first part of Figure 6 contains the ingredients of a checker for the claim $\vdash \neg C_1, \neg C_2 \Uparrow C_0$. This checking uses two constructors for indexes. The first is used to reference clauses (*i.e.*, the expression $\texttt{idx}(i)$ denotes $\neg C_i$) and the second constructor is

```
idx : int -> index                lit : form -> index
dl  : list int -> cert            ddone : cert
```

$$\forall L.\ \vee_c(\texttt{dl}(L), \texttt{dl}(L)). \qquad \forall L.\ t_e(\texttt{dl}(L)).$$
$$\forall L.\ f_c(\texttt{dl}(L), \texttt{dl}(L)). \qquad \forall L.\ \forall_c(\texttt{dl}(L), \texttt{dl}(L)).$$
$$\forall C \forall L.\ store_c(\texttt{dl}(L), C, \texttt{dl}(L), \texttt{lit}(C)). \qquad \forall L.\ \exists_e(\texttt{dl}(L), \texttt{dl}(L), T).$$
$$\forall L \forall P \forall \Theta.\ decide_e(\texttt{dl}(L), \Theta, \texttt{ddone}, \texttt{lit}(P)). \qquad \forall L.\ \wedge_e(\texttt{dl}(L), \texttt{dl}(L), \texttt{dl}(L)).$$
$$\forall I \forall \Theta.\ decide_e(\texttt{dl}([I]), \Theta, \texttt{dl}([]), \texttt{idx}(I)). \qquad \forall l \forall \Theta.\ init_e(\texttt{ddone}, \Theta, l).$$
$$\forall I \forall J \forall \Theta.\ decide_e(\texttt{dl}([I, J]), \Theta, \texttt{dl}([J]), \texttt{idx}(I)). \qquad \forall l \forall L \forall \Theta.\ init_e(\texttt{dl}(L), \Theta, l).$$
$$\forall I \forall J \forall \Theta.\ decide_e(\texttt{dl}([J, I]), \Theta, \texttt{dl}([J]), \texttt{idx}(I)). \qquad \forall L.\ release_e(\texttt{dl}(L), \texttt{dl}(L)).$$

```
rdone : cert     rlist  :         list (int * int * int) -> cert
                 rlisti : int ->  list (int * int * int) -> cert
```

$$\forall R.\ f_c(\texttt{rlist}(R), \texttt{rlist}(R)).$$
$$\forall C \forall l \forall R.\ store_c(\texttt{rlisti}(l, R), C, \texttt{rlist}(R), \texttt{idx}(l)).$$
$$t_e(\texttt{rdone}).$$
$$\forall I \forall \Theta.\ decide_e(\texttt{rlist}([]), \Theta, \texttt{rdone}, \texttt{idx}(I)) \ \texttt{:-}\ \langle \texttt{idx}(I), t \rangle \in \Theta.$$
$$\forall I, J, K, R, C, N, \Theta.\ cut_e(\texttt{rlist}([\langle I, J, K \rangle | R]), \Theta, \texttt{dl}([I, J]), \texttt{rlisti}(K, R), N) \ \texttt{:-}\ \langle \texttt{idx}(K), C \rangle \in \Theta,\ \texttt{negate}(C, N).$$

Fig. 6. Resolution certificate definition in two parts

used to index literals that need to be stored: here the literal is used to provide its own index. The first two `cert` constructors in that figure are used to control the sequencing of decide rules involving two (negated) clauses. The first of these constructors provides the sequent of clause indexes (at most 2) used to build a proof and the second constructor is used to signal that the proof should finish with the selection of stored literals and not with additional clauses.

The clerks for this part of the checking process do essentially no computation and just move certificates around unchanged: the exception is the store clerk that provides the trivial index $\texttt{lit}(C)$ for the literal C. The only expert that provides information to guide proof reconstruction is the decide expert which transforms the choice of clauses to consider from two to one to none. Given these clerks and experts, it is now the case that if C_i and C_j resolve to yield C_k then $\texttt{dl}([i, j]) \vdash \neg C_1, \ldots, \neg C_m \Uparrow C_k$ is provable. With only small changes, the binary resolution checker can be extended to hyperresolution: in this case, the experts will need to attempt to find a proof of decide depth $n + 1$ when attempting to resolve together $n \geq 2$ clauses.

To describe a checker for a complete certificate, we use three additional constructors for certificates as well as the additional clauses in the second part of Figure 6. Notice that the decide expert only proposes a focus at the end of the checking process when the list of triples (resolvents) is empty: this expert only succeeds if one of the clauses is t (the negation of the empty clause). It is the cut expert that is responsible for looping over all the triples encoding resolvents.

Notice that the cut-formula is the clause C_k and that the left premise invokes the resolvent checking mechanism described above. The right premise of the cut carries with it an index (in this case, k) so that the next step in the proof checking knows which index to use to correctly store that formula. The *LKF* proof that is implicitly built during the checking of a resolution contains one cut rule for every resolvent triple in the certificate.

4.4 Capturing General Computation within Proofs

The line between computation and deduction is certainly movable and one that a flexibly designed proof certificate definition should allow to be moved. As we saw in Section 4.1, we can use naive proof reconstruction to compute, for example, the conjunctive normal form of a propositional formula. We can go further, however, and allow for arbitrary Horn clause programs to be computed on first-order terms during proof reconstruction. For example, if one needs to check a proof rule that involves a premise that requires one number to divide another number, it is an easy matter to write a (pure) Prolog program that computes this binary relationship on numbers. Such Horn clauses can be added to the sequent context and a proof certificate could easily guide the construction of a proof of that premise from such clauses.

5 Adequacy of Encoding

Our use of the augmented *LKF* proof system as our kernel guarantees soundness no matter how the clerk and expert predicates are defined. On the other hand, one might want to know if the checker is really checking the proof intended in the certificate. A checker for a mating could, in fact, ignore the mating and run the decision procedure from Section 4.1 instead. The kernel itself cannot guarantee the *adequacy* of the checking: knowledge of the certificate definition is necessary to ensure that. As our examples show, however, the semantics of the clerk and expert predicates is clearly given by the LKF^a proof system and certificate definitions are compact: thus, verifying certificates should be straightforward.

Some aspects of a proof certificate are not possible to check using our kernel. Consider defining a *minimal proof mating* to be a proof mating for which no mated pairs can be removed and still remain a proof mating. We see no way to capture this minimality condition: that is, we see no way to write a certificate definition that successfully approves a mating if and only if it is a minimal proof mating. A similar observation can be made with resolution: if $\vdash \neg C_1, \neg C_2 \Uparrow C_0$ has a proof (even a proof of decide depth 3) it is not necessarily the case that C_0 is the resolvent of C_1 and C_2. For example, the resolution of $\forall x[p(x) \vee r(f(x))]$ and $\forall x[\neg p(f(x)) \vee q(x)]$ is $\forall x[r(f(f(x))) \vee q(x)]$. At the same time, it is possible to prove the sequent

$$\vdash \exists x[\neg p(x) \wedge \neg r(f(x))], \exists x[p(f(x)) \wedge \neg q(x)] \Uparrow \forall x[r(f(f(f(x)))) \vee q(f(x)) \vee s(f(x))].$$

This formula is similar to a resolvent except it uses a unifier that is not most general and it has an additional literal. Thus, when this check succeeds, what is checked is its soundness and not its technical status of being a resolvent.

6 The More General Kernel

As we have mentioned, a more general kernel for proof checking is based not on *LKF* but the *LKU* proof system [16]. Instead of the two polarities in *LKF*, there are four polarities in *LKU*: the polarities -1 and $+1$ denote positive and negative polarities of linear logic while the polarities -2 and $+2$ denote the positive and negative polarities of classical logic. Intuitionistic logic use formulas that have subformulas of all four polarities. In order to restrict the *LKU* proof system to emulate, say, *LKF* or *LJF*, one simply needs to describe certain restrictions to the structural rules (store, decide, release, and init) of *LKU*. The logic definition documents (see Section 1) declare these restrictions.

The *LKU* proof system makes it possible to use the vocabulary for structuring checkers in *LKF* (clerks, experts, store, decide, release) to also design checkers in the intuitionistic focused framework *LJF*. The main subtleties with using *LKU* is that we must deal with a linear logic context: since such contexts must be *split* into two contexts occasionally, some of the expert predicates need to describe which splitting is required. We have defined certificate definitions for simple and dependent typed λ-calculus: that is, the *LKU* kernel can check natural deduction proofs in propositional and first-order intuitionistic logic (de Bruijn numerals make a natural index for store/decide).

7 Related and Future Work

The first mechanical proof checker was de Bruijn's Automath [8] which was able to check significant mathematical proofs. As we have mentioned in Section 3, another early proof checker was the ML implementation of LCF's tactics and tacticals [12] (for a λProlog implementation of these, see [18]). As the number and scope of mechanical theorem proving systems has grown, so too has the need to have one prover rely on other provers. For example, the OpenTheory project [14] aims at having various HOL theorem provers share proofs. Still other projects attempt to connect SAT/SMT systems with more general theorem provers, *e.g.*, [3,6,9]. In order for prover A to not blindly trust proofs from prover B, prover B may be required to generate a certificate that demonstrates that it has formally found a proof. Prover A will then need to check the correctness of that certificate. In this way, prover A only needs to check individual certificates and not rely on trusting the whole of prover B. Of course, every pair of communicating provers could involve certificates of different formats and different certificate checker. Our goal here is to base such certificates on *foundational* and *proof-theoretic* principles and to describe programmable checkers that are guaranteed to be sound. Also, since that checker is based on well understood and well explored declarative concepts (*e.g.*, sequent calculus, unification, and backtracking search), that checker can be given many different implementations.

The Dedukti proof checker [5] implements $\lambda\Pi$ *modulo*, a dependently typed λ-calculus with functional rewriting. Given a result of Cousineau & Dowek [7] that any functional Pure Type System can be encoded into $\lambda\Pi$ modulo, Dedukti

can check proofs in such type systems. As we have described above, the proof certificate setting described here allows one to capture both dependently typed λ-terms and computations (not just functional computations). As a result, we should be able to design, following [7], proof certificates for pure type systems. The dependently typed λ-calculus LF has recently been extended to LFSC [20] and to $\mathrm{LF}_{\mathcal{P}}$ [13] so that various kinds of computations can be treated by the type checker instead of being explicitly detailed within the typed λ-term itself. Such proof objects should similarly be captured in our setting.

Getting provers to trust each other's proofs using the techniques described in this paper will require the development and acceptance of an infrastructure and associated tools, something that can clearly take time. One area where proof certificates can make an early impact is in theorem proving competitions. In such competitions, theorem provers should not be trusted but rather the proof certificates that they emit should be checked. In that case, our framework for foundational proof certificates can provide a clear semantics for what constitutes a proof certificate.

Besides the proof certificates definitions that we have described above, we have designed other examples (including proof nets for multiplicative linear logic and Frege proofs) and plan to develop more. This work on foundational proof certificates is part of a more ambitious project to design proof certificates that also allow for induction and coinduction: such certificates should allow model checkers and inductive theorem provers to communicate with each other. We also hope to eventually allow counterexamples to be checked and to interact with (partial) proofs [17].

We have only considered the problem of communicating and checking formal proofs between machines. Of course, proofs are important to humans as well. Given the fact that proof certificates can be elaborated into a *LKU* sequent proof, it might well be possible to use proof-theoretic results to construct tools that allow humans to browse and interact with formal proofs in order to learn from them. We leave such considerations for future work.

8 Conclusion

In a world where proof certificates can be designed flexibly and given precise semantics and where proof checkers can be given a high degree of trust, the sharing of proofs should become "feature zero" for all new theorem provers. That is, implementers looking to get their provers accepted broadly will need to first consider how to communicate their proof evidence as a checkable certificate. In such a world, proofs can be liberated from the technologies that produced them (*e.g.*, Coq, Isabelle, and Mizar) and can be seen as the universal and eternal objects logicians and proof theorists have long been working to place at the foundations of mathematics and computer science.

Acknowledgments. We thank Jean Pichon, Thanos Tsouanas, and the reviewers for their comments on an earlier draft of this paper. This work was funded by the ERC Advanced Grant ProofCert.

References

1. Andreoli, J.-M.: Logic programming with focusing proofs in linear logic. J. of Logic and Computation 2(3), 297–347 (1992)
2. Andrews, P.B.: Theorem-proving via general matings. J. ACM 28, 193–214 (1981)
3. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of SAT/SMT solvers to Coq through proof witnesses. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer, Heidelberg (2011)
4. Baelde, D.: Least and greatest fixed points in linear logic. ACM Trans. on Computational Logic 13(1) (April 2012)
5. Boespflug, M., Carbonneaux, Q., Hermant, O.: The $\lambda\Pi$-calculus modulo as a universal proof language. Proof Exchange for Theorem Proving, 28–43 (2012)
6. Böhme, S., Weber, T.: Designing proof formats: A user's perspective. Proof eXchange for Theorem Proving, 27–32 (August 2011)
7. Cousineau, D., Dowek, G.: Embedding pure type systems in the lambda-pi-calculus modulo. In: Della Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 102–117. Springer, Heidelberg (2007)
8. de Bruijn, N.G.: Reflections on Automath. In: Nederpelt, R.P., Geuvers, J.H., de Vrijer, R.C. (eds.) Selected Papers on Automath. Studies in Logic and the Foundations of Mathematics, vol. 133, pp. 201–228. North-Holland (1994)
9. Fontaine, P., Marion, J.-Y., Merz, S., Nieto, L.P., Tiu, A.: Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 167–181. Springer, Heidelberg (2006)
10. Gentzen, G.: Investigations into logical deduction. In: Szabo, M.E. (ed.) The Collected Papers of Gerhard Gentzen, pp. 68–131. North-Holland (1969)
11. Girard, J.-Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
12. Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Language Design and Programming Methodology. LNCS, vol. 78. Springer (1979)
13. Honsell, F., Lenisa, M., Liquori, L., Maksimovic, P., Scagnetto, I.: LF$_{\mathcal{P}}$: a logical framework with external predicates. In: LFMTP 2012: Proceedings of the Seventh International Workshop on Logical Frameworks and Meta-Languages, Theory and Practice, pp. 13–22. ACM, New York (2012)
14. Hurd, J.: The OpenTheory standard theory library. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 177–191. Springer, Heidelberg (2011)
15. Liang, C., Miller, D.: Focusing and polarization in linear, intuitionistic, and classical logics. Theoretical Computer Science 410(46), 4747–4768 (2009)
16. Liang, C., Miller, D.: A focused approach to combining logics. Annals of Pure and Applied Logic 162(9), 679–697 (2011)
17. Miller, D.: A proposal for broad spectrum proof certificates. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 54–69. Springer, Heidelberg (2011)
18. Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge University Press (June 2012)
19. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform proofs as a foundation for logic programming. Annals of Pure and Applied Logic 51, 125–157 (1991)
20. Stump, A.: Proof checking technology for satisfiability modulo theories. In: Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP) (2008)

Computation in Real Closed Infinitesimal and Transcendental Extensions of the Rationals

Leonardo de Moura[1] and Grant Olney Passmore[2]

[1] Microsoft Research, Redmond
leonardo@microsoft.com
[2] LFCS, Edinburgh and Clare Hall, Cambridge*
grant.passmore@cl.cam.ac.uk

Abstract. Recent applications of decision procedures for nonlinear real arithmetic (the theory of real closed fields, or RCF) have presented a need for reasoning not only with polynomials but also with transcendental constants and infinitesimals. In full generality, the algebraic setting for this reasoning consists of *real closed transcendental and infinitesimal extensions of the rational numbers*. We present a library for computing over these extensions. This library contains many contributions, including a novel combination of Thom's Lemma and interval arithmetic for representing roots, and provides all core machinery required for building RCF decision procedures. We describe the abstract algebraic setting for computing with such field extensions, present our concrete algorithms and optimizations, and illustrate the library on a collection of examples.

1 Overview and Related Work

Decision methods for nonlinear real arithmetic are essential to the formal verification of cyber-physical systems and formalized mathematics. Classically, these decision methods operate over the *theory of real closed fields* (RCF), the first-order theory of the reals with addition, multiplication and equality and inequality predicates. RCF is decidable, admits quantifier elimination, and a variety of mature (though necessarily worst-case hyperexponential) decision procedures exist for it. Much research has gone into making RCF decision procedures practical, especially for restricted classes of formulas commonly arising in applications.

In recent years, it has become apparent that the classical approach underlying most complete RCF methods, the 'real algebraic number approach,' is insufficient for many applications. This approach has its roots in the most influential strand of RCF decision procedure research, the theory of *cylindrical algebraic decomposition* (CAD) [2,3]. In CAD and related techniques, one makes crucial use of the following observation: Since RCF is a complete theory with $\langle \mathbb{R}, +, *, <, 0, 1 \rangle$ as a model, then, when implementing an RCF decision procedure, one is free to compute over *any* RCF while still being sure that the resulting computations

* Grant Passmore was supported by the EPSRC [EP/I011005/1 and EP/I010335/1] and by NSF EXPEDITION CNS-0926181.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 178–192, 2013.

are valid over $\mathbb{R}$. This is important from a computational point of view, as $\mathbb{R}$ is uncountable with uncomputable basic operations.

In the classical approach, instead of working over $\mathbb{R}$, one works over the *real algebraic numbers* $\mathbb{R}_{alg}$, the subfield of $\mathbb{R}$ consisting of real numbers that are roots of univariate polynomials with integer coefficients. This structure is a countable real closed field with computable basic operations, and thus provides a logically sufficient computational substructure for making RCF decisions.[1] Note, though, that this field contains no *transcendental* elements such as π or e. Indeed, a real number is transcendental precisely when it is not algebraic.

On the one hand, this lack of transcendental elements seems logically inconsequential and even computationally desirable, as transcendentals are undefinable over RCF and almost all of them are uncomputable. However, various new applications have given rise to a need for computing in real closed fields containing transcendentals. This need is especially apparent when one considers cyber-physical systems [19,1,8]. In this setting, one needs to reason about ODEs which govern the continuous dynamics of a mixed discrete-continuous system. Solving for such trajectories gives rise to arithmetical constraints involving both the standard RCF operations and the constant e. If these ODEs occur in the context of aircraft maneuvers with angular positions, then one often needs to reason also with π. Similar combinations of RCF with transcendental constants arise in mainstream efforts in formalized mathematics, such as Thomas Hales's Flyspeck project, where many inequalities of this form await verification [15].

Finally, in addition to RCFs containing common transcendental constants, there is also a need for computing in RCFs containing *infinitesimals*. This stems from applications as well, albeit indirectly: as RCF is computationally infeasible, many researchers have focused on developing decision procedures for restricted but practically useful fragments of the theory. In two of the most useful fragments, the $\exists$ and $\exists\forall$ fragments, novel decision methods have been developed which rely on infinitesimals [5,14,7]. Some of these procedures, such as the singly exponential Grigor'ev-Vorobjnov $\exists$RCF method, are also of immense theoretical interest. However, many of them have never been implemented. The lack of a viable library for computing with real closed fields containing infinitesimals has been an impediment to this line of research. A robust library providing the computational substructure for reasoning in such real closed fields would remove a serious obstruction to work on nonlinear real arithmetic, allowing decision procedure researchers to focus on higher-level concerns, especially on novel decision methods which can then rely on this foundational library as a black-box.

In this paper, we present a library for computing in real closed fields containing computable transcendental, infinitesimal and algebraic elements. In particular, our library supports computing over real closed transcendental and infinitesimal extensions of the rational numbers. This is realized through the theory of *real closures*, a classical technique in real algebraic geometry which allows one to construct new real closed fields from arbitrary ordered fields.

[1] In fact, $\mathbb{R}_{alg}$ is the *prime model* of RCF, which means $\mathbb{R}_{alg}$ isomorphically embeds into every real closed field. In this sense, $\mathbb{R}_{alg}$ is the "smallest" real closed field.

Our main contribution is threefold:

1. We show how real closed fields containing computable transcendental, infinitesimal and algebraic elements can all be constructed and computed in using a single uniform method. This includes a novel approach to representing algebraic elements which combines a classical result in real algebraic geometry known as Thom's Lemma with modern interval arithmetic.
2. We develop the abstract algebra in a concrete algorithmic manner, and present several optimizations we have devised over the naive methods. These optimizations have been vital to making our library practical.
3. We describe how researchers can immediatly make use of our library, make it available for download (source included), and give examples designed to help the decision procedure researcher easily get started in this area.

Related Work. The combination of transcendental constants and infinitesimals with nonlinear real arithmetic has been explored in many ways.

In the MetiTarski prover for transcendental inequalities [1,8], of which the second author of this paper is a coauthor, transcendentals are approximated to a fixed accuracy using families of algebraically-expressible upper and lower bounds. This approach does not integrate the transcendentals into the RCF field arithmetic, but rather uses a combination of resolution theorem proving and the algebraic bounds to reduce the proof of an inequality involving transcendentals to a sequence of pure RCF decisions (taken over $\mathbb{R}_{alg}$). This is sufficient for many applications, especially for classes of engineering problems with non-tight inequalities, but fails when the bounds are insufficient, especially when equality reasoning is needed. Modifications to MetiTarski which make use of our library should allow for much more powerful reasoning with transcendental constants, especially with regards to proving tight inequalities and identities.

A very different incorporation of transcendental constants has been taken by the interval constraint propagation (ICP) community, as exemplified by the tools RealPaver [13], RSolver [20], iSat [10] and dReal [12]. Their methods are incomplete even for ∃RCF, but are extremely effective in some classes of applications. Unlike complete RCF decision methods, their foundations are not based upon real closed field arithmetic, but rather on computing with interval approximations to field values. This arithmetic can be very efficient and is always sound, but it comes at the cost of the so-called *interval dependency problem*. This commonly gives rise to the over-approximation of intervals and is a source of incompleteness. Our work combines interval approximations with exact techniques stemming from real algebraic geometry, yielding a library for exact computations suitable for building complete RCF decision methods.

Major research on reasoning with infinitesimals has been done both in the ACL2 [11] and Isabelle/HOL [9] proof assistants. These efforts have focused on a particular class of real closed fields, the Hyperreals, which provide the basis for the nonstandard analysis (NSA) approach to differential calculus. In NSA, the Hyperreals are used mainly to justify the consistency of the NSA axioms, especially the crucial Transfer Axiom, and one does not compute with their elements

in the same way that one does when working over a countable, computable real closed field like $\mathbb{R}_{alg}$ during RCF decision procedures. In particular, the Hyperreals are a wildly uncomputable structure which depend on the choice of a non-principal ultrafilter over $\mathbb{N}$.[2] Thus, though both works involve real closed fields containing infinitesimals, our goals and approaches are very different.

The closest work to ours is that of Rioboo [21]. In this work, a library for computation in the real closure of a single infinitesimal extension of an ordered field was built within the computer algebra system Axiom. Though we share many goals and some high-level aspects of the approach (in particular, the explicit use of field towers as in the generic real closure method of Ligatsika, Rioboo and Roy [17]), our work is very different. First, the Rioboo infinitesimal methods revolve around a representation of algebraic elements using Puiseux series, a generalization of power series allowing fractional exponents. We take a completely different approach, combining Thom's Lemma with interval arithmetic. Second, Rioboo's library does not support extensions involving transcendental constants such as π and e. Making use of intervals in the root representation is crucial to the way we treat transcendental constants. Finally, his implementation only supports the use of a single infinitesimal, though he discusses supporting multiple infinitesimals as future work. Our library supports multiple transcendentals, infinitesimals and algebraic elements simultaneously.

Finally, let us mention the groundbreaking work of Coste-Roy and its later refinements which showed how Thom's Lemma could be used algorithmically to represent algebraic elements over arbitrary real closed fields, even those containing infinitesimals [6,18,3]. We build our root representation upon this work, combining their derivative sign condition chains with interval arithmetic techniques, resulting in novel root isolation and sign determination methods.

2 Theoretical Background

An ordered field is a field equipped with a total order $\leq$ upon its elements s.t.

$$\forall xyz\,[(x \leq y \;\Rightarrow\; x+z \leq y+z) \;\wedge\; (0 \leq x \;\wedge\; 0 \leq y \;\Rightarrow\; 0 \leq x*y)].$$

Both $\mathbb{Q}$ and $\mathbb{R}$ with their respective orderings are ordered fields. The complexes cannot be made into an ordered field as neither $\sqrt{-1} \leq 0$ nor $0 \leq \sqrt{-1}$ is consistent with the ordered field axioms. A field is *real closed* iff it is an ordered field with two additional properties: First, positive elements are squares

$$\forall x\,\big(0 \leq x \Rightarrow \exists y(x = y^2)\big),$$

and second, all polynomials of odd degree have a root. This latter property is expressed using an axiom scheme, with one axiom for each $n \in \mathbb{N}$:

$$\forall a_0 a_1 \ldots a_{2n} \exists z\,\big(z^{2n+1} + a_{2n} z^{2n} + \ldots + a_1 z + a_0 = 0\big).$$

[2] Note that it is consistent with ZF that *no* non-principal ultrafilters on $\mathbb{N}$ exist. Thus, to build a Hyperreal field, an uncomputable choice principle is needed. The wild uncomputability is contributed both from $\mathbb{R}$ and the ultrapower construction.

Observe that $\mathbb{R}$ and $\mathbb{R}_{alg}$ are real closed but $\mathbb{Q}$ is not.

Let $\mathbb{K}_1, \mathbb{K}_2$ be fields s.t.

$$\mathbb{K}_1 = \langle K_1, +_{\mathbb{K}_1}, *_{\mathbb{K}_1}, -_{\mathbb{K}_1}, 0_{\mathbb{K}_1}, 1_{\mathbb{K}_1}\rangle \text{ and } \mathbb{K}_2 = \langle K_2, +_{\mathbb{K}_2}, *_{\mathbb{K}_2}, -_{\mathbb{K}_2}, 0_{\mathbb{K}_2}, 1_{\mathbb{K}_2}\rangle$$

and $K_1 \subset K_2$. If the function symbols of $\mathbb{K}_1$ and $\mathbb{K}_2$ agree over all elements of K_1 (i.e., $0_{\mathbb{K}_1} = 0_{\mathbb{K}_2}$, $1_{\mathbb{K}_1} = 1_{\mathbb{K}_2}$, $\forall x, y \in K_1(x +_{\mathbb{K}_1} y = x +_{\mathbb{K}_2} y)$, and so on), then we say that $\mathbb{K}_1$ is a *subfield* of $\mathbb{K}_2$, that $\mathbb{K}_2$ is an *extension field* of $\mathbb{K}_1$, and that $\mathbb{K}_2/\mathbb{K}_1$ (pronounced "$\mathbb{K}_2$ over $\mathbb{K}_1$") is a *field extension*.[3] When no confusion should arise, we use $\mathbb{K}_1 \subset \mathbb{K}_2$ to indicate that $\mathbb{K}_1$ is a subfield of $\mathbb{K}_2$.

If $\mathbb{K}_1 \subset \mathbb{K}_2$ and $S \subset K_2$, then $\mathbb{K}_1(S)$ denotes the smallest subfield of $\mathbb{K}_2$ extending $\mathbb{K}_1$ and containing S. If $\varsigma \in K_2$ then $\mathbb{K}_1(\varsigma)$ denotes $\mathbb{K}_1(\{\varsigma\})$. We say that $\mathbb{K}_1(S)$ is the result of *adjoining* the elements of S *to* $\mathbb{K}_1$. An extension of the form $\mathbb{K}_1(\varsigma)/\mathbb{K}_1$ is called *simple*. If ς is the root of a polynomial in $\mathbb{K}_1[x]$, then the extension $\mathbb{K}_1(\varsigma)/\mathbb{K}_1$ is *algebraic*, and ς is *algebraic over* $\mathbb{K}_1$. Otherwise, the extension is *transcendental*, and ς is *transcendental over* $\mathbb{K}$. We can iterate the process of taking simple extensions so as to obtain non-simple ones, i.e., $\mathbb{K} \subset \mathbb{K}(\varsigma_1) \subset (\mathbb{K}(\varsigma_1))(\varsigma_2)$. We write $\mathbb{K}(\varsigma_1, \varsigma_2)$ for $(\mathbb{K}(\varsigma_1))(\varsigma_2)$. In this case, $\mathbb{K}(\varsigma_1, \varsigma_2)/\mathbb{K}(\varsigma_1)$ is simple, but $\mathbb{K}(\varsigma_1, \varsigma_2)/\mathbb{K}$ is not. A (finite or infinite) sequence of extensions $\mathbb{K}_1 \subset \mathbb{K}_2 \subset \ldots \subset \mathbb{K}_n \subset \ldots$ is called a *field tower*.

Example 1. $\mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$ is a field tower, and from it we can deduce that $\mathbb{Q}$ and $\mathbb{R}$ are both subfields of $\mathbb{C}$. However, the extension $\mathbb{C}/\mathbb{Q}$ is very different than the extension $\mathbb{C}/\mathbb{R}$. In particular, to obtain $\mathbb{C}$ from $\mathbb{Q}$ one must adjoin uncountably many elements, while to obtain $\mathbb{C}$ from $\mathbb{R}$ one need only adjoin $\sqrt{-1}$.

Let us now examine the process of field adjunction. Consider the field $\mathbb{Q}(\sqrt{2})$ resulting from adjoining $\sqrt{2}$ to $\mathbb{Q}$. Since $\sqrt{2}$ is a root of the polynomial $x^2 - 2 \in \mathbb{Q}[x]$, the extension $\mathbb{Q}(\sqrt{2})/\mathbb{Q}$ is algebraic. How can we build this field extension? Since a field is closed under its arithmetic operations, we know that as $\sqrt{2} \in \mathbb{Q}(\sqrt{2})$, then, for instance, $1/\sqrt{2}$, $23/\sqrt{2} + 1/2$, and $11/(3 * \sqrt{2})$ all must be in $\mathbb{Q}(\sqrt{2})$ as well. This suggests that we consider elements of $\mathbb{Q}(\sqrt{2})$ to be formal ratios of elements of the polynomial ring $\mathbb{Q}[\sqrt{2}]$, where $\sqrt{2}$ is taken to be a symbolic indeterminate subject to the constraint that $\sqrt{2} * \sqrt{2} = 2$. In fact, the situation is more subtle than this, and we describe it algorithmically in Sec. 3.3.

For transcendental extensions, we also make use of ratios of polynomials. If τ is transcendental over $\mathbb{K}$, then $\mathbb{K}(\tau)$ is isomorphic to $\mathbb{K}(x)$, where $\mathbb{K}(x)$ is the field of fractions of the polynomial ring $\mathbb{K}[x]$, i.e., the field of formal rational functions drawn from $\{p(x)/q(x) \mid p(x), q(x) \neq 0 \in \mathbb{K}[x],\ gcd(p(x), q(x)) = 1\}$. This isomorphism holds because as τ is transcendental, it has no nontrivial algebraic relationships with the elements of $\mathbb{K}$.

Let us return now to ordered fields. If $\mathbb{K}$ is an ordered field, then the subfields and extension fields of $\mathbb{K}$ we are interested in are those which respect $\mathbb{K}$'s ordering relation, i.e., $\mathbb{K}_1 \subset \mathbb{K} \subset \mathbb{K}_2$ implies $\leq_{\mathbb{K}_1} \subset \leq_{\mathbb{K}} \subset \leq_{\mathbb{K}_2}$. Any extension or subfield of an ordered field that we consider in this paper shall be of this form.

[3] Note that this field extension notation using "/" is purely formal, and does not imply a quotient structure or anything along those lines. See Ex. 1 for an example use.

If $\mathbb{K}$ is an ordered field, then it is possible to adjoin an *infinitesimal* element ϵ to $\mathbb{K}$ by treating $\mathbb{K}(\epsilon)/\mathbb{K}$ as a transcendental extension whose ordering extends the ordering of $\mathbb{K}$ subject to the following constraint:

$$\epsilon > 0 \ \wedge \ \forall k \in \mathbb{K} \ (k > 0 \ \Rightarrow \ \epsilon < k) \, .$$

We describe algorithmically the orderings in field extensions in Sec. 3.

Finally, let us turn to *real closures*. It is a fundamental result in real algebraic geometry that *every* ordered field $\mathbb{K}$ possesses a unique minimal algebraic real closed extension. This field is called the *real closure of* $\mathbb{K}$ and is written $\widetilde{\mathbb{K}}$. For example, $\widetilde{\mathbb{Q}} = \mathbb{R}_{alg}$ and $\widetilde{\mathbb{R}} = \mathbb{R}$. In this work, we are concerned with computing in real closed fields $\widetilde{\mathbb{K}}$, where $\mathbb{K}$ is obtained from $\mathbb{Q}$ by finitely many transcendental and infinitesimal extensions. In passing from $\mathbb{K}$ to $\widetilde{\mathbb{K}}$, a countably infinite collection of algebraic elements will be adjoined. However, to compute over $\widetilde{\mathbb{K}}$, we only need, at any given time, an extension of $\mathbb{K}$ by finitely many algebraic elements. $\widetilde{\mathbb{K}}$ is obtained "in the limit."[4]

To provide general support for nonlinear real arithmetic decision methods, including those methods requiring transcendentals and infinitesimals, we must provide all of the basic ordered field operations (arithmetic and ordering) over arbitrary subfields of $\widetilde{\mathbb{K}}$. The most challenging aspect occurs over proper algebraic extensions of $\mathbb{K}$, as this requires reasoning about the roots $\alpha \in \widetilde{\mathbb{K}}$ of polynomials $p \in \mathbb{K}[x]$. If $\mathbb{K}$ contains no infinitesimal elements, then these roots can always be isolated using intervals with rational endpoints. However, if $\mathbb{K}$ contains infinitesimal elements, then the situation is considerably more complicated.

3 Implementing Field Extensions

Our package implements towers of extensions beginning with $\mathbb{Q}$. Rationals are implemented as a pair of multi-precision integers. We support three kinds of extensions: transcendental, infinitesimal and algebraic. In our library, there is always a linear order $\prec$ between fields in a given tower. Consider a tower $\mathbb{Q} \subset \mathbb{Q}(\varsigma_1) \subset \ldots \subset \mathbb{Q}(\varsigma_1, \varsigma_2, \ldots, \varsigma_k)$. Then, it will hold that $\mathbb{Q}(\varsigma_1, \ldots, \varsigma_i) \prec \mathbb{Q}(\varsigma_1, \ldots, \varsigma_{i+1})$. Moreover, our field extensions must be constructed in a sequence s.t. transcendental extensions $\prec$ infinitesimal extensions $\prec$ algebraic extensions.

Abstractly, we can view a field extension as a mapping that, given an implementation for the operations of an ordered field $\mathbb{K}$, "lifts" these operations to the field extension $\mathbb{K}(\varsigma)$. In this Section, we describe how we implement these operations for each kind of extension. Let us fix some preliminaries.

Let B be the set of *binary rationals* (also known as *dyadic rationals*). B consists of rationals of the form $a/2^k$. B does not form a field, but it forms a ring and is closed under division by 2. The implementation of addition and multiplication for binary rationals is more efficient than that for rational numbers. Moreover,

[4] The situation is similar with packages for rational arithmetic. These packages allow one to compute over $\mathbb{Q}$ by constructing rationals "on demand" as they are needed. Of course, at any given time, only finitely many rationals have been constructed.

binary rationals can be normalized using just bit-shifting operations, instead of expensive integer gcd and division. Finally, division can be approximated to any precision using the same approach used in floating point arithmetic.[5] We say an interval of the form (l, u) is a B_∞-interval if $l, u \in (B \cup \{-\infty, \infty\})$ and $0 \notin (l, u)$. We produce interval approximations for all elements of our extension fields. For every non-zero element a, $\mathsf{interval}(a)$ is a B_∞-interval (l, u) containing a. Moreover, if a is not constructed using infinitesimals, then $l, u \in B$, and we then provide a procedure to refine the size of (l, u) to any desired precision. As the associated interval for a non-zero element never contains zero, the sign of any element can be read off from its associated interval. This allows us to decide comparisons between field elements by reducing the comparison of a and b to the sign of $a - b$.

3.1 Transcendental Extensions

At the bottom of our field towers, we support computable transcendental reals such as π and e. For adjoining a transcendental element τ, we require the user to provide a procedure $\mathsf{approximate}(\tau)$ s.t. given any $i \in \mathbb{N}$, $\mathsf{approximate}(\tau)(i)$ returns an open interval (l, u) s.t. $\tau \in (l, u)$ and $l, u \in B$. Moreover, the approximation must converge in the following sense: Let $\mathsf{width}(l, u) = u - l$. Then, for any k there must exist an i s.t. $\mathsf{width}(\mathsf{approximate}(\tau)(i)) \leq 1/2^k$. In our prototype, we provide implementations of $\mathsf{approximate}(\pi)$ and $\mathsf{approximate}(e)$.

When extending a field $\mathbb{K}$ with an irrational number τ, it is the user's responsibility to guarantee that τ is indeed transcendental with respect to $\mathbb{K}$. If this is not the case, then our implementation may not terminate when executing the sign determination algorithm in $\mathbb{K}(\tau)$. Note that transcendence is always relative to the field being extended, e.g., π and $\sqrt{\pi}$ are both transcendental over $\mathbb{Q}$, but $\sqrt{\pi}$ is not transcendental over $\mathbb{Q}(\pi)$ as it is a root of $x^2 - \pi \in (\mathbb{Q}(\pi))[x]$.

As discussed in Sec. 2, we represent the elements of $\mathbb{K}(\tau)$ as formal rational functions $p(\tau)/q(\tau)$, with τ treated as an indeterminate. Since τ is transcendental over $\mathbb{K}$, it is easy to check that $q(\tau)$ is not the zero polynomial by simply verifying that $q(\tau)$ is not identically zero using standard polynomial arithmetic over $\mathbb{K}[\tau]$.

The field operations for $\mathbb{K}(\tau)$ are based on polynomial arithmetic build upon the arithmetic operations of the field arithmetic for $\mathbb{K}$. We use the standard normal form of rational functions where the polynomial gcd of the numerator and denominator is one, and the denominator is a monic polynomial. In this representation, two values are equal iff they have the same normal form. The polynomial gcd is implemented using the standard Euclidean algorithm based on the polynomial remainder algorithm because it can be easily implemented for polynomials in $\mathbb{K}[x]$ when $\mathbb{K}$ is a computable field.

Example 2. Given $\frac{1}{2}\pi$, $\frac{1}{\pi+1} \in \mathbb{Q}(\pi)$, their sum is equal to $\frac{\frac{1}{2}\pi^2+\frac{1}{2}\pi+1}{\pi+1}$.

[5] Note that we can view binary rationals as arbitrary precision floating point numbers with negative exponents.

The approximating interval of size $1/2^k$ for a non-zero element $a = p(\tau)/q(\tau)$ is computed using interval arithmetic. Our procedure keeps refining the interval approximations for τ and the coefficients of $p(\tau)$ and $q(\tau)$ until the resulting interval for a does not contain zero and has width $\leq 1/2^k$. It is easy to see that this procedure always terminates when τ is transcendental over $\mathbb{K}$.

Since the approximating interval does not contain zero, we can use it to infer the sign of any element of $\mathbb{K}(\tau)$. Moreover, we can decide whether $a < b$ by computing the sign of $a - b$. For efficiency in the actual implementation, we first try to compare a and b using their approximating intervals. If the intervals do not overlap, then we can answer the query by simply comparing the lower and upper bounds of the intervals. Otherwise, we refine the approximating intervals until they do not overlap or their size is smaller than a user-provided threshold. If the threshold is reached, then we compute the sign of $a - b$.

3.2 Infinitesimal Extensions

An infinitesimal extension $\mathbb{K}(\epsilon)/\mathbb{K}$ adjoins a new infinitesimal to $\mathbb{K}$. Our implementation supports an arbitrary number of infinitesimals. Each new infinitesimal is infinitely smaller than any previously added infinitesimal. Note that every infinitesimal is also transcendental with respect to $\mathbb{K}$. Because of this, we also use formal rational functions to represent the elements of $\mathbb{K}(\epsilon)$. It then suffices to present the interval machinery we use to compute the ordering relation.

Note that $1/\epsilon$ is larger than any element of $\mathbb{K}$. We say $1/\epsilon$ is an *infinite value*. The initial interval approximation for ϵ is the interval $(0, 1/2^{k_\epsilon})$, where $k_\epsilon \in \mathbb{N}$ is a user-specified parameter. Thus, the initial interval approximation for $1/\epsilon$ is $(2^{k_\epsilon}, \infty)$. We say intervals of the form $(-\infty, u)$ and (l, ∞) are *non-refinable*. Only elements constructed using infinitesimals may have non-refinable intervals.

Given a non-zero polynomial $p(\epsilon)$ of the form $a_n\epsilon^n + \ldots + a_1\epsilon + a_0$ with $a_0 \neq 0$, the approximating interval of width $1/2^k$ for $p(\epsilon)$ is the approximating interval of width $1/2^k$ for a_0. If $a_0 = 0$, we say $p(\epsilon)$ is *infinitesimal*. Consider $p(\epsilon)$ infinitesimal and let a_i be the first non-zero coefficient. If a_i is negative, then the approximating interval for $p(\epsilon)$ is $(-1/2^{k_\epsilon}, 0)$, otherwise it is $(0, 1/2^{k_\epsilon})$.

Let $k \in \mathbb{K}(\epsilon)$ s.t. $k = (a_n\epsilon^n + \ldots + a_1\epsilon + a_0)/(\epsilon^m + \ldots + b_1\epsilon + b_0) \neq 0$. If $a_0 \neq 0$ and $b_0 \neq 0$, then an approximating interval of size $1/2^k$ is computed by refining the intervals for a_0 and b_0 until the desired precision is reached. If either a_0 or b_0 is non-refineable, then k is also non-refinable. Note that we never have $a_0 = 0$ *and* $b_0 = 0$, since in this case the numerator and denominator can be simplified by dividing them by ϵ. Thus, if $a_0 = 0$, we must have $b_0 \neq 0$, and k is an infinitesimal value. If $b_0 = 0$, then $a_0 \neq 0$, and k is an infinite value. Note that even when we cannot refine an approximating interval for k, we can still compute a B_∞-interval containing k.

3.3 Algebraic Extensions

At the top of our towers sit algebraic extensions. Recall that an algebraic extension $\mathbb{K}(\alpha)/\mathbb{K}$ is obtained by adjoining to $\mathbb{K}$ a root α of a polynomial $p \in \mathbb{K}[x]$.

Given such an extension, we call p the *defining polynomial of* α. Note that the algebraic extensions of $\mathbb{K}$ that we support are always subfields of $\widetilde{\mathbb{K}}$. To represent elements of $\mathbb{K}(\alpha)$, we need to be able to compute with roots of p which reside in the real closure $\widetilde{\mathbb{K}}$. *Thom's Lemma* is a classical result in real algebraic geometry which guarantees that we can always distinguish the roots of a polynomial over a real closed field (even those containing infinitesimals) using only the signs of its derivatives [6]. We base our representation upon this fact, and introduce a number of enhancements. Due to space limitations, we are forced to only present the most salient aspects of how we compute in algebraic extensions.[6]

Sign assignments. Given a set of polynomials Q, a *sign assignment* S is a mapping from Q to $\{-1, 0, 1\}$. To improve readability, we shall represent S using sets of atoms. For example, $\{q_1 \mapsto -1, q_2 \mapsto 0\}$ is represented as $\{q_1 < 0, q_2 = 0\}$. We identify a root α of a polynomial p using a pair consisting of an open B_∞-interval and a sign assignment for a subset of the derivatives of p. The sign assignment stores the sign of these derivatives at α. If $\mathbb{K}$ does not contain infinitesimal extensions, then the sign assignments are not necessary for distinguishing roots. For example, $\sqrt{2}$ can be encoded as $(x^2 - 2, (1, 2), \{\})$, i.e., as the only root of $x^2 - 2$ within $(1, 2)$ satisfying the empty sign assignment.

Example 3. Let $\mathbb{Q}(\epsilon)/\mathbb{Q}$ be an infinitesimal extension. The three roots of the polynomial $\epsilon^2 x^5 - \epsilon x^3 - \epsilon x^2 + 1 \in (\mathbb{Q}(\epsilon))[x]$ can be encoded as

$$\begin{array}{ll} (\epsilon^2 x^5 - \epsilon x^3 - \epsilon x^2 + 1, (-\infty, 0), & \{\}) \\ (\epsilon^2 x^5 - \epsilon x^3 - \epsilon x^2 + 1, (0, \infty), & \{60\epsilon^2 x^2 - 6\epsilon > 0\}) \\ (\epsilon^2 x^5 - \epsilon x^3 - \epsilon x^2 + 1, (0, \infty), & \{60\epsilon^2 x^2 - 6\epsilon < 0\}) \end{array}$$

We need a sign assignment to distinguish the two positive roots because they are bigger than any real number, and cannot be isolated using an interval. In the example above, the sign of the third derivative was used to distinguish between these two roots. Recall that *Thom's Lemma* guarantees that we can always distinguish the roots of a polynomial over an RCF using only the signs of its derivatives. Since the interval contains only two roots that need to be discriminated, we only need to find *one* derivative that has a different sign for each root. We use the third derivative, because it is the lowest degree derivative which discriminates the two roots. For example, the fourth derivative $120\epsilon^2 x$ is not used because it is positive for both roots. Later, we show how the *Sign Determination Algorithm* is used to compute the signs of these derivatives. This example also demonstrates that we often need only a proper subset of the derivatives.

Square-free polynomials. We say a polynomial $p \in \mathbb{K}[x]$ is *minimal* if p does not contain a non-trivial factor. A polynomial is *square-free* if it is does not have roots with multiplicity greater than 1. Note that given a square-free polynomial p and an interval (a, b) that contains only one root of p s.t. a and b are not roots of p, it follows that $\mathsf{sign}(p(a)) = -\mathsf{sign}(p(b))$, where sign is a function that

[6] Further details shall be in an expanded version. See Sec. 4 for a source code URL.

maps a value into the set $\{-1, 0, 1\}$. Given a polynomial p, we define the function $\mathsf{sqf}(p) = p/\mathsf{gcd}(p, p')$, where p' is the first derivative of p. It is well-known that $\mathsf{sqf}(p)$ is square-free with the same roots as p. WLOG, let us now consider polynomials of the form $a_n x^n + \ldots + a_0$ s.t. $a_0 \neq 0$. In our representation, we do not require defining polynomials to be minimal because polynomial factorization over extension fields is an expensive operation. Instead, we use square-free polynomials because they are faster to be computed, and more importantly, as the Intermediate Value Theorem holds over every real closed field, we can refine the (refinable) interval (a, b) containing a single root α by just computing the sign of p at the midpoint m: If $\mathsf{sign}(p(a)) = \mathsf{sign}(p(m))$, then the new interval is (m, b). If $\mathsf{sign}(p(b)) = \mathsf{sign}(p(m))$, then the new interval is (a, m). In the very unlikely case when $\mathsf{sign}(p(m)) = 0$, then α is actually the binary rational m.

Polynomial remainder sequences and Sturm-Tarski. Let $\mathsf{quo}(q, p)$ and $\mathsf{rem}(q, p)$ denote the polynomial quotient and remainder (resp.) of $q, p \in \mathbb{K}[x]$, i.e., $q = \mathsf{quo}(q, p) \cdot p + \mathsf{rem}(q, p)$ s.t. $\mathsf{deg}(\mathsf{rem}(q, p)) < \mathsf{deg}(p)$, where $\mathsf{deg}(p)$ is the degree of p. If p is the defining polynomial for α, then $p(\alpha) = 0$ and consequently $q(\alpha) = \mathsf{rem}(q, p)(\alpha)$. This allows us to use polynomial remainders to simplify any $q(\alpha)$. The *signed polynomial remainder* $\mathsf{srem}(q, p)$ is defined as $-\mathsf{rem}(q, p)$. A Sturm polynomial sequence $[s_1; s_2; \ldots; s_k]$ for polynomials p and q is defined inductively as $s_1 = p$, $s_2 = q$, $s_i = \mathsf{srem}(s_{i-2}, s_{i-1})$, where $\mathsf{srem}(s_{k-1}, s_k) = 0$. We use $\mathsf{sturm}(p, q)$ to denote the Sturm polynomial sequence for p and q. Given a sequence S of polynomials in $\mathbb{K}[x]$, we use $\mathsf{sv}(S, a)$ for the number of sign changes (ignoring zeroes) in the sequence when each polynomial is evaluated at a. For example, $\mathsf{sv}([2+x^2+x^3;\ 2x+3x^2;\ -18+2x;\ -1], 0) = 1$, since there is only one sign variation in the sequence evaluated at 0. We use $\mathsf{pos}(q, p, a, b)$, $\mathsf{neg}(q, p, a, b)$ and $\mathsf{zero}(q, p, a, b)$ to denote the number of roots β of p s.t. $\beta \in (a, b)$ and $q(\beta)$ is positive, negative and zero respectively. The *Sturm-Tarski Theorem* states that given a polynomial sequence $S = \mathsf{sturm}(p, q \cdot p')$, it holds that[7]

$$\mathsf{sv}(S, a) - \mathsf{sv}(S, b) = \mathsf{pos}(q, p, a, b) - \mathsf{neg}(q, p, a, b).$$

Following Basu-Pollack-Roy [3], we define a *Tarski Query* $\mathsf{TaQ}(q, p; a, b)$ as

$$\mathsf{TaQ}(q, p; a, b) = \mathsf{sv}(\mathsf{sturm}(p, q \cdot p'), a) - \mathsf{sv}(\mathsf{sturm}(p, q \cdot p'), b)$$

and remark that

$$\begin{aligned}
\mathsf{TaQ}(1, p; a, b) &= \mathsf{zero}(q, p, a, b) + \mathsf{pos}(q, p, a, b) + \mathsf{neg}(q, p, a, b),\\
\mathsf{TaQ}(q, p; a, b) &= \mathsf{pos}(q, p, a, b) - \mathsf{neg}(q, p, a, b),\\
\mathsf{TaQ}(q^2, p; a, b) &= \mathsf{pos}(q, p, a, b) + \mathsf{neg}(q, p, a, b).
\end{aligned}$$

Moreover, $\mathsf{TaQ}(1, p; a, b)$ is the number of roots of p in the interval (a, b). If α is the only root of p in the interval (a, b), then the sign of $q(\alpha)$ can be determined using $\mathsf{TaQ}(q, p; a, b)$.

[7] The Sturm-Tarski Theorem is actually for half-open intervals of the form $(a, b]$. WLOG, we assume that for any root α encoded as $(p, (a, b), S)$, b is not a root of p. If it is, we encode α as $(p/(x - b), (a, b), S)$ instead.

Sign determination. Tarski Queries are also used to implement the *Sign Determination Algorithm* [3]. Given a set of polynomials $Q = \{q_1, \ldots, q_k\}$, $\mathsf{signdet}(Q, p, a, b)$ returns the feasible sign assignments of Q at the roots of p in the interval (a, b). Actually, it computes more than that: for each sign assignments S, it returns the number of roots of p in (a, b) that satisfy S. For example, for $Q = \{q\}$, $\mathsf{signdet}$ can compute the feasible sign assignments by computing $\mathsf{TaQ}(1, p; a, b)$, $\mathsf{TaQ}(q, p; a, b)$ and $\mathsf{TaQ}(q^2, p; a, b)$ and solving the system of equations above. For $Q = \{q_1, q_2\}$, in the worst case, we have to compute $\mathsf{TaQ}(h, p; a, b)$ for each h in the set $\{1, q_2, q_2^2, q_1, q_1q_2, q_1q_2^2, q_1^2, q_1^2q_2, q_1^2q_2^2\}$. In general for a set $Q = \{q_1, \ldots, q_k\}$, $\mathsf{signdet}$ will, in the worst case, have to compute 3^k Tarski Queries for polynomials of the form $\prod_{q \in Q, i \in \{0,1,2\}} q^i$, and solve a system of 3^k equations. We implement a more efficient $\mathsf{signdet}$ of Ben-Or *et al.* [4].

Now, assume that α is encoded as $(p, (a, b), S)$, and we want to determine the sign of $q(\alpha)$. We can decide that by computing $R = \mathsf{signdet}(\mathsf{poly}(S) \cup \{q\}, p, a, b)$, where $\mathsf{poly}(S)$ is the set of polynomials occurring in S. Then,

$$\begin{array}{l} \mathsf{if}\ S \cup \{q = 0\} \in R\ \mathsf{then}\ q(\alpha) = 0, \\ \mathsf{if}\ S \cup \{q > 0\} \in R\ \mathsf{then}\ q(\alpha) > 0, \\ \mathsf{if}\ S \cup \{q < 0\} \in R\ \mathsf{then}\ q(\alpha) < 0. \end{array}$$

We know that one and only one of the cases above can be true because p has only one root in the interval (a, b) satisfying the sign conditions S.

To compute an upper-bound for the positive roots of $(\sum_{i=0}^{n} a_i x^i) \in \mathbb{K}[x]$, we use Knuth's bound $2(\mathsf{max}\{\sqrt[k]{(-a_{n-k}/a_n)} \mid 1 \leq k \leq n, a_{n-k} < 0\})$. As the a_i may be neither integer nor rational values, we estimate $(-a_{n-k}/a_n)$ using the approximating intervals for a_{n-k} and a_n. Let s be the upper bound of the resulting interval. If the upper-bound for s is ∞, then so is the sought upper-bound. Otherwise, we compute the least integer j s.t. $s \leq 2^j$. The value j can be easily computed based on the bit-wise log_2 operation for integers. Finally, we approximate the kth root as $2^{\frac{j}{k}+1}$. Note that if $\mathbb{K}$ does not contain infinitesimals, then the computed upper-bound is a binary rational of the form 2^m. The lower-bound for a positive root is computed by computing the upper-bound for $x^n p(1/x)$. For negative roots, we compute the bounds for the positive roots of $p(-x)$.

Clean representations. We represent elements of $\mathbb{K}(\alpha)$ as polynomials $q(\alpha)$. We define inductively the predicate $\mathsf{clean}(a)$. If $a \in \mathbb{Q}$, then $\mathsf{clean}(a)$ holds if a is an integer. If a is an element of a transcendental or infinitesimal extension $\mathbb{K}(\varsigma)$, then $\mathsf{clean}(a)$ holds if a is of the form $p(\varsigma)/1$ and for all coefficients c of $p(\varsigma)$, $\mathsf{clean}(c)$ holds. Similarly, if a is an element of an algebraic extension, then $\mathsf{clean}(a)$ holds if a is represented by a polynomial with clean coefficients. When $\mathsf{clean}(a)$ holds, we say a is *clean*. In our experiments, we observed that minimizing the use of gcd (especially with non-clean elements) is by far the most important optimization. Many operations with clean elements produce clean elements, and consequently do not require expensive normalization operations based on gcd.

Let a and b be clean elements of transcendental or infinitesimal extensions. Then, $a + b$, $-a$ and $a \cdot b$ are also clean. We recall that for algebraic extensions

$\mathbb{K}(\alpha)$, an element a is represented as a polynomial $q(\alpha)$, and this polynomial can be (optionally) simplified to $\mathsf{rem}(q, p)$, where p is the defining polynomial for α. If p is monic with clean coefficients and a is clean before applying the simplification, it will remain clean after applying it. Unfortunately, this is not the case for non-monic polynomials. For example, $\mathsf{rem}(\alpha^3 + 1, 3\alpha^2 - 1) = 1 + (2/3)\alpha$.

To minimize the generation of non-clean elements, we generate Sturm sequences using polynomial pseudo-remainders. We use $\mathsf{pquo}(q, p)$ and $\mathsf{prem}(q, p)$ to denote the polynomial pseudo-quotient and pseudo-remainder of $q, p \in \mathbb{K}[x]$, and remark that $l^d q = \mathsf{pquo}(q, p) \cdot p + \mathsf{prem}(q, p)$, where l is the leading coefficient of p, and d is the number of iterations used to compute $\mathsf{pquo}(q, p)$ and $\mathsf{prem}(q, p)$. The signed pseudo remainder is defined as

$$\mathsf{sprem}(q, p) = \begin{cases} \mathsf{prem}(q, p), & \text{if } l < 0 \wedge d \text{ is odd} \\ -\mathsf{prem}(q, p), & \text{otherwise} \end{cases}$$

The main motivation for this is that for any element a, $\mathsf{sign}(\mathsf{sprem}(q, p)(a)) = \mathsf{sign}(\mathsf{srem}(q, p)(a))$. Because for Tarski Queries only the number of sign alternations matter, we can use sprem instead of srem when generating Sturm sequences.

Given a polynomial p with clean coefficients, if we disable the algebraic normalizations using non-monic defining polynomials, and compute Sturm sequences using sprem, then all elements in the generated sequence are clean. With this approach, we observed a dramatic performance improvement (cf. Sec. 4).

Now, let us show how we represent elements of $\mathbb{K}(\alpha)$ as polynomials even when the defining polynomial p for α is not minimal. When p is minimal, given a non-zero element a represented using a polynomial $q(\alpha)$, we can represent $1/a$ using a polynomial $h(\alpha)$. Let r be $\mathsf{rem}(q, p)$. As p is minimal, $\gcd(p, r) = 1$. Then, using the extended gcd algorithm we can compute polynomials h and g such that $g \cdot p + h \cdot r = 1$ Since, $p(\alpha) = 0$ and $r(\alpha) = q(\alpha)$, we have $h(\alpha) \cdot q(\alpha) = 1$. If p is not minimal, the $\gcd(p, r)$ may be different from 1, with h the zero polynomial. To cope with this problem, we simply replace the defining polynomial p for α with $(p/\gcd(p, r))$ whenever $\gcd(p, r) \neq 1$.

Root isolation. Finally, we summarize all the steps used in our root isolation procedure for polynomials $p \in \mathbb{K}[x]$. First, we make sure that 0 is not a root of p, p is square-free and has clean coefficients. Then, we estimate the lower and upper bounds for positive and negative roots. WLOG, we focus on the positive case. If the upper-bound is ∞, we use the sign determination procedure for distinguishing the positive roots using the signs of the derivatives of p. If the upper bound is not ∞, we try to isolate the roots using interval bisection and binary search. If $\mathbb{K}$ does not depend on infinitesimals, the procedure always terminates. If $\mathbb{K}$ depends on infinitesimals, to guarantee termination, we interrupt the binary search if the size of the interval in a branch is smaller than a user provided parameter, and switch to the approach based on sign determination. Note that if p does not depend on infinitesimal values, then more efficient root isolation methods can be used [23].

4 Examples

In this section, we present a small set of examples using our package. Our library was implemented as a module in the Z3 theorem prover[8], and we provide a C API and Python bindings.

Introductory examples. We demonstrate the basic capabilities of our package using the Python bindings. A polynomial is described as a list of coefficients. `MkRoots` returns the roots of a polynomial as a list. In the following command we consider the roots of $x^2 - 2$. In Python, `**` is the power operator.

```
msqrt2, sqrt2 = MkRoots([-2, 0, 1])
print(sqrt2)
>> root(x^2 + -2, (0, +oo), {})
print(1/sqrt2)
>> 1/2*root(x^2 + -2, (0, +oo), {})
print(sqrt2**2 == 2)
>> True
print(sqrt2.decimal(10))
>> 1.4142135623?
print(sqrt2**3 + 1)
>> 2*root(x^2 + -2, (0, +oo), {}) + 1
```

The procedure `MkInfinitesimal` creates a new infinitesimal extension, while `Pi` and `E` return π and e respectively. In the following example, we extract the first (and only) root of the polynomial $x^3 + \epsilon\, x^2 + (\sqrt{2} + \pi)\, x - \pi$.

```
eps = MkInfinitesimal("eps")
pi  = Pi()
r   = MkRoots([-pi, sqrt2 + pi, eps, 1])[0]
print(r)
>> root(x^3 + eps*x^2 + (root(x^2 + -2, (0, +oo), {}) + pi)*x +
        -1*pi, (0, +oo), {})
print(r.decimal(10))
>> 0.6337173142?
```

Now, we show basic computations with infinitesimals and transcendentals. First, we compare $2 + 2\pi + \pi^2 - 2\epsilon - 2\pi\epsilon + \epsilon^2 < 2 + 2\pi + \pi^2$, and then we compare ϵ and $\sqrt[3]{\epsilon}$.

```
print(2 + 2*pi + pi**2 - 2*eps - 2*pi*eps + eps**2 < 2 + 2*pi + pi**2)
>> True
eps3 = MkRoots([-eps, 0, 0, 1])[0]
print(eps3 > eps)

>> True
print(1/eps > 10000000000000000000000000000)
>> True
```

The examples above are stored in the file basic.py.

[8] The code is available at `http://z3.codeplex.com/wikipage?title=CADE24`. Experiments were done on an Intel Core i7-2620 2.7Ghz CPU with 8Gb RAM, and the package was compiled using the GMP multi-precision library.

MetiTarski. MetiTarski uses the nlsat [16] nonlinear solver in Z3. Real algebraic number computations are often a bottleneck for nlsat, and were implemented using the textbook approach of polynomials with integer coefficients and an interval with binary rational endpoints. The following example was extracted from a MetiTarski/Z3 execution trace where nlsat times out after 30 min. In this example, nlsat assigns the first root of the following polynomial to variable x.

$$216\,x^{15} + 4536\,x^{14} + 31752\,x^{13} - 520884\,x^{12} - 42336\,x^{11} - 259308\,x^{10} + 3046158\,x^9 + 140742\,x^8 + 756756\,x^7 - 5792221\,x^6 - 193914\,x^5 - 931392\,x^4 + 3266731\,x^3 + 90972\,x^2 + 402192\,x + 592704$$

Then, it replaces x with the assigned value in the polynomial $y^3 + x^3 + 1$, and timeouts trying to isolate the roots of the result. In our package presented in this paper, these two operations are performed in 0.05 secs (nlsat.py).

Tower of extensions. In this example, we create a tower of extensions containing two transcendental (π and e), one infinitesimal (ϵ), and 5 algebraic extensions. The algebraic extensions are the first roots of the following 5 polynomials:

$$\begin{aligned}
r_0 &:= x^4 + -2\,\epsilon\,x^3 + (\epsilon^2 - 4)\,x^2 + 4\epsilon\,x + 4 - 2\epsilon^2 + 4\\
r_1 &:= (-\epsilon^6 + 8\epsilon^4 - 20\epsilon^2 + 16)r_0 - 8\epsilon^5 + 32\epsilon^3 - 32\epsilon + (2\epsilon^4 - 8\epsilon^2 + 8)x^2\\
r_2 &:= x^5 + 3\,x^3 + r_1\,x^2 - 1\\
r_3 &:= x^5 + r_1\,x^3 + \pi\,r_2\,x^2 - 3\\
r_4 &:= 8\,x^5 + r_3\,x^4 + e\,x^2 + x - 7
\end{aligned}$$

All roots are isolated in 0.28 secs (tower8.py). However, if we do not use clean representations (cf. Sec. 3), it takes 31 secs for the 4th polynomial, and times out after 30 min in the last one (tower8_eager_norm.py).

Rioboo examples [21]. All solved in a negligible amount of time (rioboo*.py).

Strzebonski examples [22]. All solved in a negligible amount of time (strz.py).

5 Conclusion

We have presented a library for computing in real closed transcendental and infinitesimal extensions of the rationals. This provides a computational substructure sufficient for implementing many advanced (and hitherto practically unexplored) decision methods for nonlinear real arithmetic. We hope that both the library and the ideas underlying it will prove useful to the community, and that it may become the foundation of new practically useful decision methods.

References

1. Akbarpour, B., Paulson, L.C.: Applications of MetiTarski in the Verification of Control and Hybrid Systems. In: Majumdar, R., Tabuada, P. (eds.) HSCC 2009. LNCS, vol. 5469, pp. 1–15. Springer, Heidelberg (2009)
2. Arnon, D.S., Collins, G.E., McCallum, S.: Cylindrical Algebraic Decomposition. I. The Basic Algorithm. SIAM J. Comp. 13(4), 865–877 (1984)

3. Basu, S., Pollack, R., Roy, M.: Algorithms in Real Algebraic Geometry. Springer, Secaucus (2006)
4. Ben-Or, M., Kozen, D., Reif, J.: The complexity of elementary algebra and geometry. In: STOC. ACM (1984)
5. Canny, J.: Some algebraic and geometric computations in PSPACE. In: Twentieth ACM Symposium on Theory of Computing, STOC. ACM (1988)
6. Coste, M., Roy, M.: Thom's lemma, the coding of real algebraic numbers and the computation of the topology of semi-algebraic sets. JSC 5(1-2) (1988)
7. de Moura, L., Passmore, G.O.: Exact nonlinear optimization on demand. In: Preparation (2013)
8. Denman, W., Akbarpour, B., Tahar, S., Zaki, M.H., Paulson, L.C.: Formal verification of analog designs using MetiTarski. In: FMCAD, pp. 93–100 (2009)
9. Fleuriot, J.D., Paulson, L.C.: A Combination of Nonstandard Analysis and Geometry Theorem Proving, with Application to Newton's Principia. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS (LNAI), vol. 1421, pp. 3–16. Springer, Heidelberg (1998)
10. Fränzle, M., Herde, C., Ratschan, S., Schubert, T., Teige, T.: Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. JSAT 1, 209–236 (2007)
11. Gamboa, R., Kaufmann, M.: Nonstandard Analysis in ACL2. JAR 27(4) (2001)
12. Gao, S., Avigad, J., Clarke, E.M.: δ-complete decision procedures for satisfiability over the reals. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 286–300. Springer, Heidelberg (2012)
13. Granvilliers, L., Benhamou, F.: RealPaver: An Interval Solver using Constraint Satisfaction Techniques. ACM Trans. on Maths. Software 32, 138–156 (2006)
14. Grigor'ev, D.Y., Vorobjov Jr., N.N.: Solving systems of polynomial inequalities in subexponential time. JSC 5(1-2), 37–64 (1988)
15. Hales, T.C., Harrison, J., McLaughlin, S., Nipkow, T., Obua, S., Zumkeller, R.: A revision of the proof of the Kepler conjecture. Discrete & Computational Geometry 44(1), 1–34 (2010)
16. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 339–354. Springer, Heidelberg (2012)
17. Ligatsikas, Z., Rioboo, R., Roy, M.: Generic computation of the real closure of an ordered field. Maths. and Comp. in Sim. 42(4-6), 541–549 (1996)
18. Mishra, B., Pedersen, P.: Arithmetic with real algebraic numbers is in NC. In: ISSAC 1990, pp. 120–126. ACM, New York (1990)
19. Platzer, A., Clarke, E.M.: Formal verification of curved flight collision avoidance maneuvers: A case study. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 547–562. Springer, Heidelberg (2009)
20. Ratschan, S.: Efficient Solving of Quantified Inequality Constraints over the Real Numbers. ACM Trans. on Comp. Logic 7(4), 723–748 (2006)
21. Rioboo, R.: Infinitesimals and real closure. Technical report, Laboratoire D'Informatique de Paris 6 (2001)
22. Strzebonski, A.: Computing in the field of complex algebraic numbers. JSC 24(6) (1997)
23. Strzeboński, A., Tsigaridas, E.P.: Univariate real root isolation in multiple extension fields. In: ISSAC 2012, pp. 343–350. ACM, New York (2012)

A Symbiosis of Interval Constraint Propagation and Cylindrical Algebraic Decomposition*

Ulrich Loup[1], Karsten Scheibler[2], Florian Corzilius[1], Erika Ábrahám[1], and Bernd Becker[2]

[1] RWTH Aachen University, Germany
[2] University of Freiburg, Germany

Abstract. We present a novel decision procedure for non-linear real arithmetic: a combination of `iSAT`, an incomplete SMT solver based on interval constraint propagation (ICP), and an implementation of the complete cylindrical algebraic decomposition (CAD) method in the library `GiNaCRA`. While `iSAT` is efficient in finding unsatisfiability, on satisfiable instances it often terminates with an interval box whose satisfiability status is unknown to `iSAT`. The CAD method, in turn, always terminates with a satisfiability result. However, it has to traverse a double-exponentially large search space.

A symbiosis of `iSAT` and CAD combines the advantages of both methods resulting in a fast and complete solver. In particular, the interval box determined by `iSAT` provides precious extra information to guide the CAD-method search routine: We use the interval box to *prune the CAD search space* in both phases, the projection and the construction phase, forming a search "tube" rather than a search tree. This proves to be particularly beneficial for a CAD implementation designed to search a satisfying assignment pointedly, as opposed to search and exclude conflicting regions.

1 Introduction

The formal modeling of systems and their properties along with corresponding analysis and synthesis methods require the usage of appropriate logics. In addition, many algorithms from this area need decision procedures for *satisfiability checking*, i.e., algorithms to decide whether there exists an assignment of values to the variables occurring in a formula such that the formula evaluates to true. A typical example is bounded model checking [4], a technique to encode counterexamples of a certain length by formulas; a solution to such a formula provides a counterexample, which can be used for the correction of the erroneous system.

In this context, propositional logic with *SAT solving* as a decision procedure is widely used for discrete systems. For more complex systems more expressive logics, e.g., fragments of first-order logic over some theories, are necessary.

* This work has been partially supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "AVACS" (SFB/TR 14, `http://www.avacs.org/` and the Research Training Group "AlgoSyn" (GRK 1298, `http://www.algosyn.rwth-aachen.de/`)

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 193–207, 2013.

Satisfiability-modulo-theories (*SMT*) *solving* turned out to be a very successful technique, which combines SAT solving with theory decision procedures: the Boolean structure of a formula is handled by a SAT solver, whereas the consistency of sets of theory atoms is checked by a *theory solver*. In the last decade a lot of effort has been put into the development of efficient SMT solvers for, e.g., equality logic with uninterpreted functions and linear real arithmetic.

Recently, increasing interest is devoted to solvers for *quantifier-free non-linear real arithmetic* (*QFNRA*). However, available SMT solvers for this expressive, highly challenging logic are rare. One of the reasons is the complexity of the available methods for checking sets of polynomial constraints over the reals for consistency. This circumstance complicates an embedding into an *efficient* SMT-solving framework. `Z3` [12] and `iSAT` [9] are among the most prominent SMT solvers for QFNRA. `Z3` uses an elegant adaption of the *cylindrical algebraic decomposition* (*CAD*) method and is complete for QFNRA. `iSAT` is based on the efficient technique of *interval constraint propagation* (*ICP*), yielding a fast but, in its current version, incomplete tool.

In this paper we introduce an extension of `iSAT` with an adapted variant of the CAD method, turning `iSAT` into a practically efficient and complete SMT solver for QFNRA. Because of the high complexity of solving polynomial constraints, we implement a *full-lazy* interaction between ICP and CAD. The `iSAT` algorithm recursively splits the initially bounded search space into smaller boxes and applies ICP to reduce the box sizes by cutting down provably unsatisfying parts. Under certain conditions, `iSAT` can detect that all points in a box are solutions, or else, that no solutions are in a box. In all other cases, `iSAT` continues to split the respective box. To assure termination, `iSAT` stops this splitting process when the box size reaches a lower threshold. This is the point when we invoke our CAD solver: to decide whether the remaining box contains a satisfying solution.

In turn, we use such a box to restrict the CAD search space. This could easily be formulated by extending the original constraints with constraints representing the bounds to the variables as given by the box. However, this would only complicate the CAD computation. Instead, we adapt the CAD method itself to be able to use the given bounds to prune the search. To this end, we implement approaches similar to the ones proposed in [11], but kept simple enough for efficiency maintenance. The CAD method consists of two phases, the *projection* and the *construction* phase. For the projection phase, we propose a novel pruning operator. The composition of our pruning operator and Hong's improved projection operator [10] generalizes Hong's projection operator as well as the model-based projection operator introduced in [12]. For the construction phase, we modify the procedure for the computation of CAD cells to consider only those cells which have a non-empty intersection with the given box. These modifications together allow us to take full advantage of the interval boxes of `iSAT` to reduce the size of the CAD remarkably. Our tool and the benchmarks we used are available at `http://ginacra.sourceforge.net/cade2013.html`.

Besides the related approaches [12,11] already mentioned, we are aware of [14,1,16], where approximation together with some form of validation is used to speed up the CAD computation.

2 Preliminaries

We use $\mathbb{Z}$ to denote the set of integers and $\mathbb{N}$ to denote the set of natural numbers including 0. We use the notation $]a,b[= \{c \in \mathbb{R} \mid a < c < b\}$ for open intervals, $[a,b] = \{c \in \mathbb{R} \mid a \leq c \leq b\}$ for closed intervals and define half-open intervals analogously. Furthermore, we permit unbounded open and half-open intervals by using ∞ or $-\infty$ as bounds. $\mathbb{I}_{\mathbb{R}}$ denotes the set of all intervals in $\mathbb{R}$. We call $I_1 \times \cdots \times I_n \in {\mathbb{I}_{\mathbb{R}}}^n$ an *interval box* composed of the intervals $I_j \in \mathbb{I}_{\mathbb{R}}$, $1 \leq j \leq n$. Given an interval $I \in \mathbb{I}_{\mathbb{R}}$, $\llcorner I$ denotes the *lower* or *left bound* and $I^{\urcorner}$ the *upper* or *right bound* of I.

2.1 Real Arithmetic

We start with a formal definition of our input language, *quantifier-free nonlinear real arithmetic* (*QFNRA*). QFNRA *formulas* φ are Boolean combinations of *constraints* c which compare *polynomials* p to 0. A polynomial p can be a *constant*, a *variable* x, or a sum, difference or product of polynomials:

$$\begin{array}{lllllllll} p & ::= & 0 & \mid & 1 & \mid & x & \mid (p+p) & \mid (p-p) \mid (p \cdot p) \\ c & ::= & p=0 & \mid & p<0 & \mid & p>0 & & \\ \varphi & ::= & c & \mid & (\neg\varphi) & \mid & (\varphi \wedge \varphi) & & \end{array}$$

Further operators such as disjunction $\vee$, implication $\rightarrow$ etc. and the weak relations $\leq$ and $\geq$ can be defined as syntactic sugar. We define $\cdot$ to bind stronger than $+$,$-$ and $\neg$ stronger than $\wedge$ stronger than $\vee$ and sometimes omit parentheses when causing no confusion. We use the standard semantics of QFNRA formulas.

Let $p = a_1 x_1^{e_{1,1}} \cdots x_n^{e_{n,1}} + \cdots + a_k x_1^{e_{1,k}} \cdots x_n^{e_{n,k}}$ be a polynomial with $a_j \in \mathbb{Z}$ and $e_{i,j} \in \mathbb{N}$ for $1 \leq i \leq n$ and $1 \leq j \leq k$. By $\deg(p) := \max_{1 \leq j \leq k}(\sum_{i=1}^{n} e_{i,j})$ we denote the *degree of* p. A formula φ is *linear* if $\deg(p) \leq 1$ for all polynomials p in φ, and *non-linear* otherwise. We denote the set of all polynomials with integer coefficients and variables $x_1,\ldots,x_n$ for some $n \geq 1$ by $\mathbb{Z}[x_1,\ldots,x_n]$. A polynomial $p \in \mathbb{Z}[x_1,\ldots,x_n]$ is called *univariate* if $n = 1$, and *multivariate* otherwise.

2.2 Satisfiability-Modulo-Theories (SMT) Solving

In this paper, we tackle the following problem:

QFNRA Satisfiability Problem.	
Given:	QFNRA formula φ over the variables $x_1, \ldots, x_n$.
Question:	Is there an assignment $\alpha : \{x_1, \ldots, x_n\} \rightarrow \mathbb{R}$ satisfying φ?

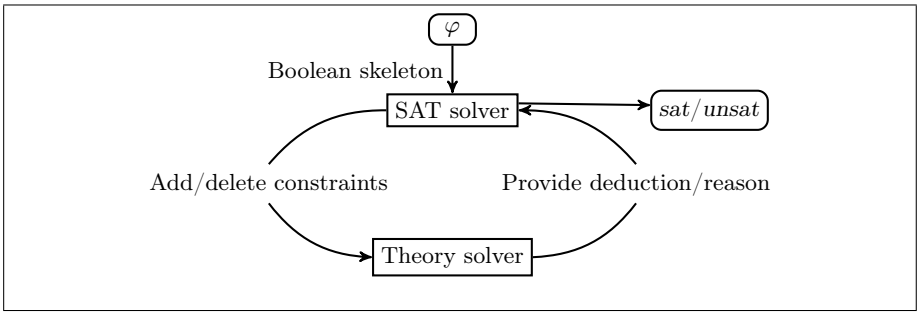


Fig. 1. Basic scheme of an SMT solver

To solve this problem we use the technique of *satisfiability-modulo-theories* (*SMT*) solving, whose basic scheme is depicted in Fig. 1.

The *Boolean skeleton* of a QFNRA formula replaces each polynomial constraint in the formula by a fresh Boolean variable, resulting in a *propositional logic* formula. The *propositional satisfiability problem* poses the question whether there exists an assignment to the propositions in such a formula rendering the formula `true`.

Most modern *SAT solvers*, offering efficient solutions to this problem, are based on the DPLL procedure [7]. The input formula is required to be in *conjunctive normal form (CNF)*, i.e., it is a conjunction of *clauses*, whereas each clause is a disjunction of *literals*, the latter being variables or their negations. The Tseitin transformation [17] can be used for an equisatisfiable transformation of propositional logic formulas into CNF with linear complexity (in the number of operators) on the cost of adding linearly many new variables.

Given an input propositional logic formula in CNF, DPLL-style SAT-solvers assign values to variables following some heuristics and apply *Boolean constraint propagation (BCP)* to detect implications of the assignments. Such implications stem from *unit* clauses with all literals but one being false, implying that the last literal must be true in order to satisfy the formula. When the propagation leads to a conflict, i.e., when all literals of a clause are false, the solver uses *conflict resolution* to derive a reason for the conflict. *Conflict-driven clause learning (CDCL)* [15] can be used to exclude this and similar conflicts from future search.

SMT solving generalizes SAT solving by allowing literals of the input formula to be atoms from some theories, in our case QFNRA constraints, or their negations. A SAT solver works on the Boolean skeleton of the underlying problem and assigns `true` or `false` to the theory atoms. The SAT solver is complemented by a *theory solver* offering a decision procedure for the conjunction of constraints from the underlying theory.

There are different approaches how to combine a theory solver and a SAT solver in an SMT application. In the *full lazy* approach, the SAT solver first searches for a complete satisfying assignment for the Boolean skeleton. If the skeleton is unsatisfiable then the input formula is also unsatisfiable. If a satisfying

assignment is found, the SAT solver invokes the theory solver to check whether the conjunction of those theory atoms, which satisfy the clauses, is consistent. If this is the case then the input formula is satisfiable. Otherwise, the Boolean skeleton is refined with a conflict clause which forbids the current combination of the conflicting theory atoms for the future search. In the *less lazy* approach, the theory solver is invoked more frequently also for partial assignments. In some other solvers, the separation between SAT and theory solving is not so strict (see Section 3). More details on SMT solving can be found, e.g., in [5, Ch.26].

3 The SMT Solver `iSAT`

In contrast to most other SMT solvers, in `iSAT` [9] the separation between the theory solver and the SAT solver is not so strict. Instead, `iSAT` tightly integrates *interval constraint propagation* (*ICP*) (see e.g. [3]) into the SAT framework. This deep integration has the advantage of sharing the common core of the search algorithms between the propositional and the theory-related part of the solver.

The `iSAT` solver allows Boolean and interval-bounded integer-valued and real-valued *variables*. Boolean assignments for propositions are extended with interval valuations $\rho : \mathit{Var} \to \mathbb{I}_{\mathbb{R}}$ assigning to each numerical variable from *Var* its current interval bound. Note that in `iSAT` all variables have initial intervals.

Beyond linear and non-linear integer and real arithmetic expressions, `iSAT` supports transcendental functions and a collection of further operators and functions. Examples for `iSAT` *theory atoms* are $x^2 + y^2 = z^2$, $|v - w| \leq \min(v, w)$ or $\sqrt[3]{x} + \sin y < e^z$. But in this paper we focus on non-linear problems only.

As a *preprocessing* step `iSAT` applies some basic arithmetic simplifications to the input formula (e.g., the expression $2x+3x$ is rewritten to $5x$). Furthermore, it tries to detect contradictions, tautologies and subsumptions (e.g., $x < 5 \wedge x < 7$ is simplified to $x < 5$, and $x < 5 \wedge x > 7$ is reduced to `false`). The preprocessed formula is then rewritten into CNF with the Tseitin transformation.

The `iSAT` algorithm operates on CNFs containing only *simple bounds* and *primitive constraints* as theory atoms. Simple bounds compare a variable to a constant. Primitive constraints are equalities containing exactly one unary or binary operator. Constraints of other forms are decomposed by a Tseitin-like satisfiability-equivalent transformation into simple bounds and primitive constraints. E.g., the theory atom $x + y^2 > 0$ is replaced by $h_1 > 0$ and the unit clauses $h_1 = x + h_2$ and $h_2 = y^2$ are added to the clause set, where h_1 and h_2 are type-consistent fresh auxiliary variables.

The three basic elements decision, propagation and conflict resolution of the DPLL framework are also present in `iSAT`, but they are extended for the operation on integer- and real-valued intervals in addition to the Boolean domain. Deciding an integer- or real-valued variable corresponds to splitting its interval into two intervals and selecting the lower or upper one.

In the propagation phase, ICP is executed in addition to BCP. ICP tries to narrow the interval bounds of numerical variables based on simple bounds

and primitive constraints. However, if a new bound has a negligible progress compared to the existing bound then the new bound is ignored to prevent infinite propagation sequences and thus to guarantee the termination of the ICP process. Furthermore, primitive constraints are checked whether they are still consistent with the current interval valuations of the variables. A primitive constraint c is *consistent* under an interval valuation iff there exist values in the intervals of the involved variables which evaluate c to true. Inconsistency can be detected when ICP deduces an empty interval for one of the involved variables.

ICP is incomplete, i.e., `iSAT` may terminate with an inconclusive answer, because in general equations like $x = y \cdot z$ can only be satisfied by point intervals. But reaching such point intervals by ICP cannot be guaranteed for real-valued variables. In such cases `iSAT` will return an interval box with a possible solution called a *candidate solution*. In [8] an approach is presented to check whether there exists a satisfying assignment in a candidate solution. Beside the fact that the approach is heuristic and thus may not succeed in all cases it is also unable to detect the unsatisfiability of an interval box. Using the CAD overcomes these two problems.

4 Cylindrical Algebraic Decomposition (CAD)

In this work, we use the *cylindrical algebraic decomposition* (*CAD*) method, introduced by Collins [6], to check the satisfiability of a set of non-linear real arithmetic constraints comparing polynomials to 0. Although the CAD method originally was designed to perform quantifier elimination, its adaption to our setting of solving constraint systems is marginal because the basic scheme of the CAD method is still the same. The CAD method decomposes the state space into a finite number of connected regions called *cells* in which the involved polynomials are sign-invariant, i.e., in which the truth values of the constraints are invariant. The method also allows to get a sample point from each of the cells. The satisfiability check of the constraint set can thus be done by checking whether one of the finitely many sample points satisfies all the constraints. The finite union of the satisfying cells builds the solution set of the given constraints.

Fig. 2 shows an example of a solution set in $\mathbb{R}^2$ for three constraints. In general, such solution sets are unions of open or closed connected subsets of $\mathbb{R}^n$ whose boundaries are defined by the real zeros of the considered polynomials over n variables. However, a CAD of the solution space is usually an even finer decomposition of $\mathbb{R}^n$, arranged in cylinders in each dimension. We describe the precise structure of a CAD by an explanation of the actual CAD computation. We use the constraint set defined in Example 1 in order to illustrate the method.

Example 1. We consider the constraint set $\{x^2 + y^2 > 4,\ x + 2 = y,\ x^2 y > 1\}$. A picture of its solution set in $\mathbb{R}^2$ is given in Fig. 2.

Let $P_n \subseteq \mathbb{Z}[x_1, \ldots, x_n]$ be a set of multivariate polynomials over $n \geq 1$ variables. The CAD method decomposes the state space $\mathbb{R}^n$ into a finite number of disjoint subsets (cells) $\cup_{i=1}^{m} R_i$ such that in each cell R_i the polynomials in P_n are *sign*

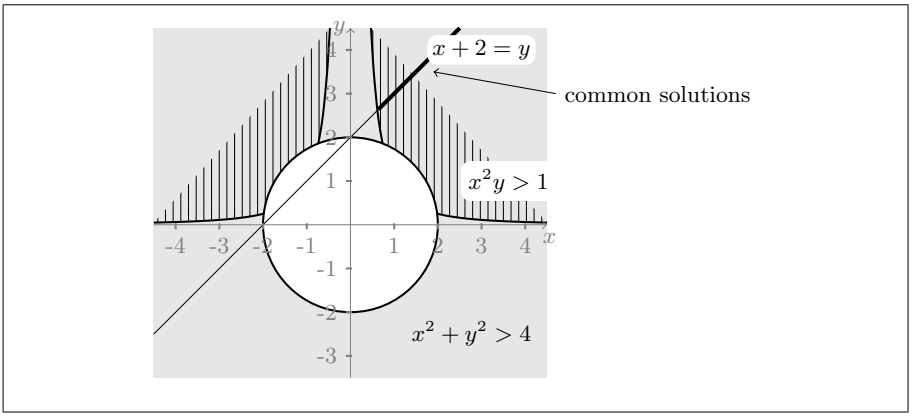


Fig. 2. Solution sets of the constraints $x^2 + y^2 > 4$ (every point outside the circle around the center with radius 2), $x + 2 = y$ (angle bisecting line) and $x^2y > 1$ (hatched area)

invariant, i.e., for all $p \in P_n$ either $p(a) > 0$, $p(a) < 0$ or $p(a) = 0$ for all $a \in R_i$. Such a CAD can be used to decide consistency of a set of constraints comparing polynomials from P_n to 0 by substituting a sample point from each of the CAD cells into the constraints. The CAD is done in two phases, the *projection phase* and the *construction phase*, as illustrated in Fig. 3.

4.1 Projection Phase

The left side of Fig. 3 depicts the projection phase. From the input set P_n, the CAD method computes another set of polynomials $\text{proj}_{x_n}(P_n) \subseteq \mathbb{Z}[x_1, \ldots, x_{n-1}]$ so that the following holds: Let $R \subseteq \mathbb{R}^{n-1}$ be a connected set such that each $p \in \text{proj}_{x_n}(P_n)$ is sign-invariant on R. Then there is a decomposition $R \times \mathbb{R} = \biguplus_{i=1}^{k} R \times S_i$ of the cylinder over R with $k \in \mathbb{N} \setminus \{0\}$ and $S_i \subseteq \mathbb{R}$ connected for $1 \leq i \leq k$ so that each $p \in P_n$ is sign-invariant on $R \times S_i$ for $1 \leq i \leq k$. We call the operation proj_{x_n} a (*CAD*) *projection operator*.

There are several implementations of projection operators in the literature. In this work, we use Hong's improved projection operator [10]. We illustrate such a CAD projection in Example 2: The given projection eliminates the variable x from the input set over the variables x and y. For a deeper insight into how the projection works, we refer to the before-mentioned articles on CAD [6,10].

Example 2 (CAD projection). We consider the polynomials corresponding to the constraint set of Example 1 $P = \{x^2 + y^2 - 4,\ x + 2 - y,\ x^2y - 1\} \subseteq \mathbb{Z}[x, y]$. Then $\text{proj}_x(P)$ contains the 3 coefficients from the input polynomials $2 - y$, y, $y^2 - 4$, but also 5 polynomials representing multiple or common roots, such as $-y^3 + 4y - 1$, $y^3 - 4y^2 + 4y - 1$, $y^6 - 8y^4 + 2y^3 + 16y^2 - 8y + 1$.

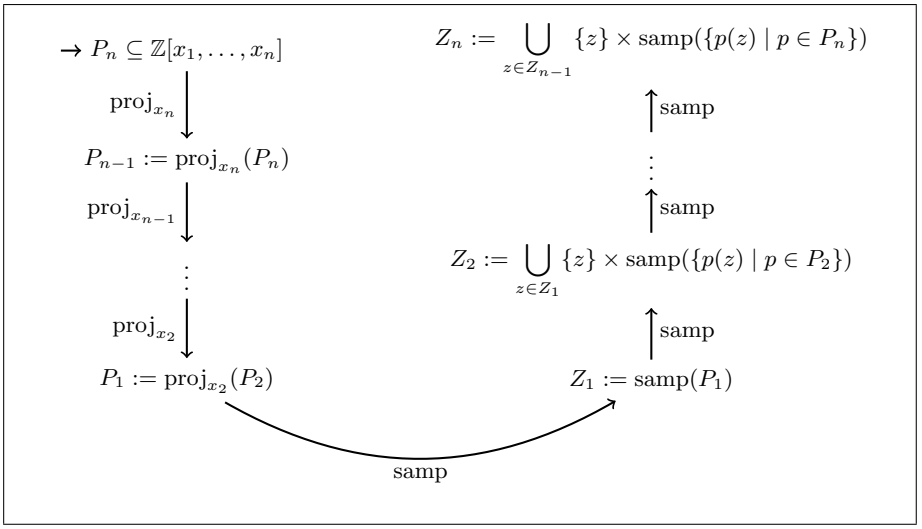


Fig. 3. The basic CAD method in a nutshell: the projection phase on the left, the construction phase on the right

We call $P_i := \text{proj}_{x_{i+1}}(P_{i+1}) \subseteq \mathbb{Z}[x_1, \ldots, x_i]$ the *ith successive projection* for $0 < i < n$. We start the construction phase with the set of univariate polynomials P_1.

4.2 Construction Phase

In the construction phase, as sketched on the right side of Fig. 3, the CAD method starts with computing the set $Z_1 \subseteq \mathbb{R}$ of representatives for the cells of a CAD of the real line. Let $r_1 < \cdots < r_k$ be the real roots of the polynomials in P_1, then the maximal sets where each $p \in P_1$ is sign-invariant are the intervals $]-\infty, r_1[$, $[r_1, r_1]$, $]r_1, r_2[$, $\ldots$, $[r_k, r_k]$, $]r_k, \infty[$. This family of intervals is a *CAD of* $\mathbb{R}$. We call a representative of such an interval (*CAD*) *sample*, i.e., samples are real roots of univariate polynomials, the points between these roots, and one point less than the smallest and one greater than the largest root. The set of samples of a set of univariate polynomials U is denoted by $\text{samp}(U)$.

The procedure to compute CAD samples is one of the most important parts of the CAD method. Hence, we shed some more light on algorithms for finding real roots of polynomials.

The heart of real root isolation is counting real roots of a univariate polynomial in a given interval. By p' we denote the formal derivative of a univariate polynomial p, and by $(p \mod q)$ the remainder of the division of the polynomials p by q. Given a univariate polynomial p, its *Sturm sequence* is $S = (p_0, \ldots, p_k)$ with $p_0 = p$, $p_1 = p'$, $p_i = -(p_{i-2} \mod p_{i-1})$ for $2 \leq i \leq k$, and p_k a rational constant. Let $S(t)$ for $t \in \mathbb{Q}$ denote the sequence of the non-zero $p_i(t)$ values, and let $\text{Var}(S(t))$ be the number of sign changes in $S(t)$. Sturm's

theorem (see e.g. [2, Theorem 2.50]) states that p has $\mathrm{Var}(S(a)) - \mathrm{Var}(S(b))$ many real roots in $]a, b[$ if a and b are no roots of p themselves. For example, the Sturm sequence of $p = x^2 - 2$ is $S = (x^2 - 2, 2x, 2)$. Therefore, p has $\mathrm{Var}(S(-2)) - \mathrm{Var}(S(2)) = 2 - 0 = 2$ real roots in $]-2, 2[$.

In this example, a radius of 2 around 0 already suffices to capture all real roots of p. To obtain such an upper bound on the radius covering all roots of a univariate polynomial $p = \sum_{i=0}^{k} c_i x^i$, $c_k \neq 0$, we can use the *Cauchy bound* $\mathrm{C}(p) := \sum_{d=0}^{k} \left|\frac{c_d}{c_k}\right|$. In order to isolate all real roots of p by intervals, the interval $]-\mathrm{C}(p), \mathrm{C}(p)[$ can be successively split, e.g. by its midpoint, until intervals are found for which Sturm's theorem determines exactly one real root. An interval containing exactly one real root r of p is called an *isolating interval for* r, and $\langle p,\ ]a, b[\ \rangle$ an *interval representation for* r. Given two interval-represented real roots r_1 and r_2, we are able to effectively compute arithmetic operations such as $r_1 + r_2$, $-r_1$, $r_1 \cdot r_2$, r_1^{-1}, as well as relations such as $r_1 = r_2$ and $r_1 < r_2$ (see, e.g., [13, p. 327ff]).

Example 3 shows possible CAD samples including interval-represented roots.

Example 3 (CAD samples). Given $\mathrm{proj}_x(P)$ from Example 2, $\mathrm{samp}(\mathrm{proj}_x(P))$ contains 19 samples, e.g., the roots $r_1 := \langle -y^3 + 4y - 1,\ ]\frac{-17}{8}, \frac{-67}{32}[\ \rangle$, $r_2 := \langle -y^3 + 4y - 1,\ ]\frac{61}{160}, \frac{31}{80}[\ \rangle$, $r_3 := \langle -y^3 + 4y - 1,\ ]\frac{31}{80}, \frac{15}{8}[\ \rangle$ of $-y^3 + 4y - 1$ and corresponding intermediate points $-3 < r_1 < -1 < r_2 < \frac{1}{3} < r_3 < \frac{15}{8}$. The isolating intervals of the real roots in $\mathrm{samp}(\mathrm{proj}_x(P))$ are even tighter because of several refinements when computing the whole example; however, we omit these details for the sake of readability.

Hence we have $Z_1 = \mathrm{samp}(P_1)$ and continue with the computation of Z_{i+1} with $Z_{i+1} := \bigcup_{(a_1,\ldots,a_i)\in Z_i} \{(a_1, \ldots, a_i, a_{i+1}) \mid a_{i+1} \in \mathrm{samp}(\{p(a_1, \ldots, a_i, x_{i+1}) \mid p(x_1, \ldots, x_{i+1}) \in P_{i+1}\})\}$ for all $1 < i < n$. Note that $p(a_1, \ldots, a_i, x_{i+1})$ is indeed a univariate polynomial. We call a point $z \in Z_n \subseteq \mathbb{R}^n$ a (*CAD*) *sample point*.

Because $\mathrm{samp}(\{p(z) \mid p \in P_{i+1}\})$ contains all samples of a CAD of $\mathbb{R}$ for each $z \in Z_i$, we can conclude by induction on i, $1 \leq i < n$, that Z_n contains sample points for all sign-invariant connected components of $\mathbb{R}^n$. Note that the calculation of $p(z)$ for given $p \in P_{i+1}$, $z \in Z_i$, $1 \leq i < n$ can be performed by computing resultants [2, Sec. 4.2].

If $m = |P|$ and d be the maximum degree of all polynomials occurring in P and if we assume that $|\mathrm{proj}(P)| \leq m^2 d^2$ as in [10], then $|Z_n|$ is dominated by $\mathcal{O}(m^{2^n-1} d^{2^n+n-1})$. This worst case can occur even for linear input formulas, as showed in [18].

5 iSAT Adaption

ICP works well for robust combinations of theory atoms, but may fail to give a conclusive answer for non-robust ones. Here, a set of theory atoms is meant to be robust, if the atoms maintain their truth value under small perturbations

of the constants in the atoms. Nonetheless even a candidate solution contains useful information which can be used as a guiding path for a complete method like the CAD method.

Whereas `iSAT` operates on a CNF enriched with primitive constraints as unit clauses, the CAD method is fed with the non-decomposed theory atoms together with the current interval valuation of the variables occurring in those atoms in a full-lazy manner. The CAD method is used whenever `iSAT` is unable to give a conclusive answer by itself. As mentioned earlier the focus of this paper are non-linear problems. Note, that we could also allow the actual input logic of `iSAT` and would then simply avoid calling the CAD method when, e.g. transcendental functions, occur. In order to determine which theory atoms have to be passed to the CAD method, each simple bound originating from a decomposed theory atom keeps a link to its original non-decomposed form.

`iSAT` would stop its search with a candidate solution, because each primitive constraint is still consistent under the current interval valuation of the variables. However, it is unknown whether there exists a point interval for each variable satisfying all primitive constraints. In such a case, having the link back from the simple bounds to the original theory atoms makes it possible to determine the set of atoms to pass to the CAD method to get a definitive answer. The solver terminates with a satisfiable result when the CAD method finds a solution within the given interval box defined by the candidate solution. Otherwise a conflict clause excluding this set of atoms is created and the search continues. Note, that `iSAT` could call the CAD method even in case of unbounded problems making the extension of `iSAT` complete for QFNRA.

6 CAD Adaption

In Section 4 we showed that the two phases of the CAD method, the projection and the construction phase, can be both extremely inefficient because of their doubly-exponential search space. However, because our CAD-method implementation is designed to traverse all sample points for a satisfying one, we can mainly avoid this worst case behavior when we assume that a possible solution lies within a given interval box.

At first, we formally fix the situation of a call to the CAD solver within the ICP solver. Let, throughout this section, $B = I_1 \times \cdots \times I_n \in {\mathbb{I}_{\mathbb{R}}}^n$ an interval box with $B \neq \emptyset$, $P = \{p_1, \ldots, p_m\}$ polynomials with $p_i \in \mathbb{Z}[x_1, \ldots, x_n]$ over $n \geq 1$ variables and $p_1 \sim_1 0, \ldots, p_m \sim_m 0$ be constraints with $\sim_i \in \{<, =, >\}$ for $1 \leq i \leq m$. We are interested in the satisfiability of

$$\bigwedge_{i=1}^{m} p_i(x_1, \ldots, x_n) \sim_i 0 \quad \wedge \quad \bigwedge_{i=1}^{n} x_i \in I_i. \tag{1}$$

The constraints representing the bounds can be formulated using the polynomials $P_B := \{x_i - b_i \mid b_i \in \{\llcorner I_i, I_i \urcorner\} \cap \mathbb{R}, 1 \leq i \leq n\}$. Note that the infinity bounds are excluded in this representation.

A straightforward approach to solve (1) is to construct a CAD of R^n w.r.t. $P \cup P_B$ and search a satisfying sample point. We refer to this solution approach as *lazy method* because the bounds are checked at the very end of the construction phase – lazily.

This section, however, is devoted to an approach we call *eager method*: the bounds are incorporated deeply into the CAD phases in order to restrict the search space w.r.t. B, as opposed to leaving it untouched and checking membership in B at the end. The adaption of the CAD method for the eager approach is two-fold: a reduction of the sets of polynomials computed in the projection phase and an early pruning of the CAD samples in the construction phase.

Projection Phase. Based on the interval box B, we define a pruning operator prun_B so that $|\text{prun}_B(P)| \leq |P|$.

Definition 1 (Interval-based polynomial pruning). *Let* $Q \subseteq \mathbb{Z}[x_1, \ldots, x_n]$. *We call the operation* $\text{prun}_B(Q) := \{p \in Q \mid p(r) = 0 \text{ for some } r \in B\}$ interval-based polynomial pruning.

The set $\text{prun}_B(P)$ consists of those polynomials from P which have a real root in the given interval box. Thus, $\text{prun}_B(P) \subseteq P$.

Note that the composition $\text{prun}_B \circ \text{proj}_{x_i}$ is an implementation of the projection operator introduced in [11], generalizing the *model-based* projection operator described in [12] where the authors prune the output of the operator using numbers, possibly interval-represented, instead of arbitrary intervals.

The polynomial pruning $\text{prun}_B(Q)$ can be straightforwardly implemented by testing whether $p \in Q$ has a root inside B by constructing a CAD. Because this approach can be very inefficient, we first compute an over-approximation of $\{p(a) \mid a \in B\}$ by interval arithmetic and check whether $(0, \ldots, 0)$ is contained. If not, we can safely prune p. Otherwise we compute a CAD w.r.t. p and prune p if no satisfying sample for $p = 0$ is found. Else, we have to keep p for the computation of the CAD of P and, as an optimization, we can reuse all projection polynomials computed for p.

We demonstrate the "tube" effect of the interval-based polynomial pruning in the projection phase by two examples depicted in Fig. 4 and Example 4.

Example 4 (Interval-based polynomial pruning). Let P be as in Ex. 2. We consider the two interval boxes B_1, $B_2 \in \mathbb{I}\mathbb{R}^2$ with $B_1 = [-3, 0]^2$ (cf. left picture in Fig. 4), $B_2 =]1, 4[\times]2, 4[$ (cf. right picture in Fig. 4) and P_{B_1}, P_{B_2} their polynomial representations each having 4 additional input polynomials.

The projection $\text{proj}_x(P \cup P_{B_1})$ has 14 elements. Fortunately, $\text{prun}_{B_1}(P \cup P_{B_1})$ already removes $x^2y - 1$ resulting in a smaller projection $\text{prun}_{B_1}(\text{proj}_x(P \cup P_{B_1}))$ of 9 elements. The projection $\text{proj}_x(P \cup P_{B_2})$ contains 18 elements and $\text{prun}_{B_2}(P \cup P_{B_2})$ leaves just $x + 2 - y$ so that $\text{prun}_{B_2}(\text{proj}_x(P \cup P_{B_2}))$ consists only of 8 linear elements.

The example shows that B restricts the set of projections in such a way that the problem can become significantly smaller, e.g., for B_2 the initial problem is reduced to a linear polynomial only.

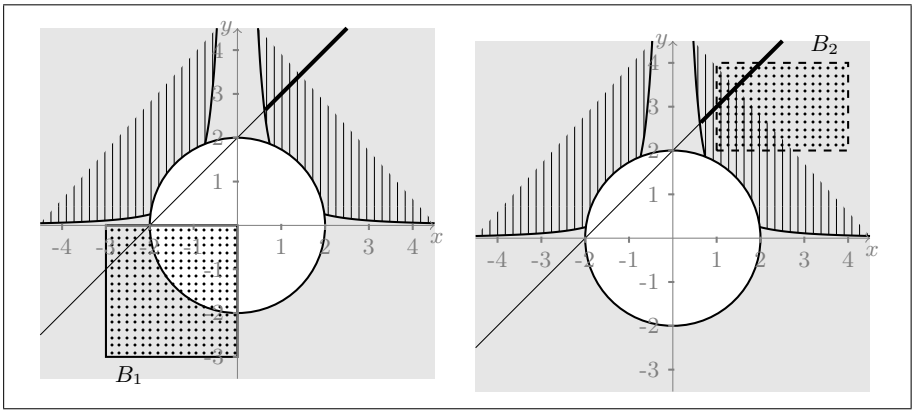


Fig. 4. Solution sets of the constraints $x^2 + y^2 > 4$, $x + 2 = y$ and $x^2 y > 1$ intersected with the boxes $B_1 = [-3, 0]^2$ and $B_2 =]1, 4[\times]2, 4[$

Construction Phase. We can use the interval box B to restrict also the sample construction for each dimension.

Definition 2 (Interval-based samples). *Let $U \subseteq \mathbb{Z}[x_i]$ be a set of univariate polynomials. We define the set* $\text{samp}_B(U)$ *by*

$$\text{samp}_B(U) := \text{samp}(U \cup P_{I_i}) \cap [\llcorner I_i, I_i \urcorner].$$

$\text{samp}_B(U)$ *is called a set of* interval-based *(*CAD*)* samples*.*

Let $U \subseteq \mathbb{Z}[x_i]$. The important aspects of $\text{samp}_B(U)$ are, first, the inclusion of the bounds as real roots in order to ensure that points between possible real roots of U and the bounds of I_i are constructed, and second, the intersection with the closed version of I_i to prune all samples outside the bounds of I_i but not the bounds themselves in order to guarantee that the bounds are available for the sample construction in higher dimensions.

In our implementation of $\text{samp}_B(U)$, we use I_i as initial interval for the real root isolation due to Sturm's theorem rather than the Cauchy bounds so that only real roots inside I_i are found. We then apply the usual CAD sample construction and remove the outer samples at the left and right bounds.

Joining of the Two Adapted Phases. The new operators from the Definitions 1 and 2 enable us to define the eager method. We depict its scheme in Fig. 5 referring to the scheme of the native CAD method in Fig. 3.

In the adapted projection phase, we use the adapted ith successive projection $P_i := \text{prun}_B(\text{proj}_{x_{i+1}}(P_{i+1} \cup P_{I_{i+1}}))$ for $0 < i < n$. In particular, we project the polynomials representing the bounds only when the corresponding variable is projected and do not keep the bound-representing polynomials themselves in the projections. Moreover, the interval-based polynomial pruning is applied after each projection and to the input set P.

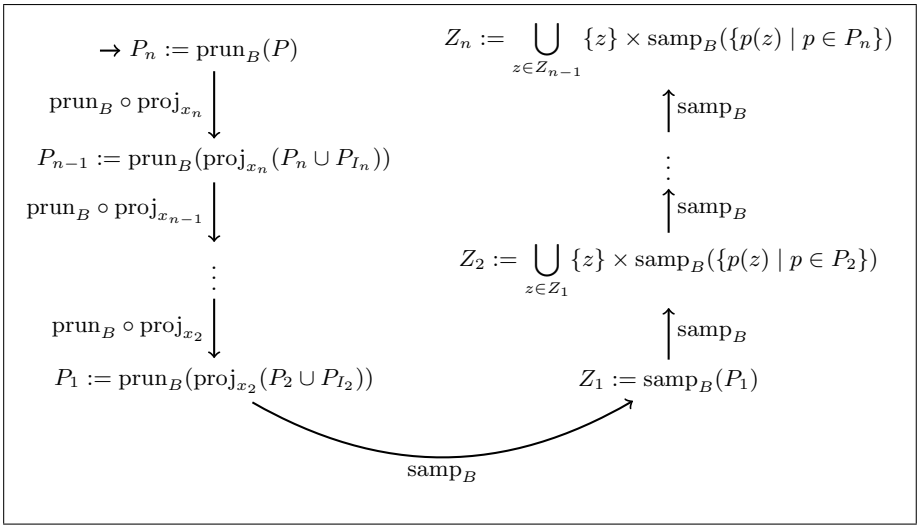


Fig. 5. Eager method to solve (1)

The adapted construction phase, in turn, is working analogously to the native construction phase as shown in Fig. 3, except that the interval-based sample operator samp_B is used.

Both phases work hand in hand. Firstly, prun_B could cut off also the polynomials in P_B which are important to construct samples between the bounds and possible real roots inside B. This shortcoming is tackled by samp_B, as it always produces the bounds of B as roots. Moreover, the projection of the bound-representing polynomials together with other polynomials have to be generated. We force this by adding the respective polynomials from P_B for the projections.

7 Experimental Results

We have embedded three versions of the CAD procedure of Section 6 into `iSAT`: $\texttt{iSAT+CAD}_1$ implementing the lazy method, $\texttt{iSAT+CAD}_2$ the eager method without interval-based polynomial pruning and $\texttt{iSAT+CAD}_3$ the eager method itself. We compared our approaches with two well-known solvers also based on CAD: `Z3` and `Redlog` (calling `rlcad`). Table 1 shows the performances of `iSAT` and our CAD implementation embedded in a DPLL-style SMT solver, `SMT+CAD`.

The tests cover the benchmark sets `hong` (Hon), `meti-tarski` (Met), `zankl` (Zan) and `keymaera` (Key) as described in [12]. In addition, we tested the set `Laser` (Las) containing unrolled bounded model checking instances which describe whether a laser can reach a target state traversing a labyrinth of reflecting obstacles.

Table 1. Results obtained on a 2.1 GHz AMD with a timeout of 200 seconds

bench (#)	Hon (20)		Met (8276)		Las (381)		Zan (166)		Key (421)		*all* (9264)	
	#	*sec*	#	*sec*	#	*sec*	#	*sec*	#	*sec*	#	*sec*
iSAT+CAD$_3$	20	0.1	7634	18840.0	331	758.6	31	195.4	331	216.4	8347	20010.5
- sat	0	0.0	4967	15678.1	0	0.0	12	183.8	0	0.0	4979	15861.7
- unsat	20	0.3	2667	3161.9	331	758.6	19	11.6	331	216.4	3368	4148.8
iSAT+CAD$_2$	20	0.3	7321	29143.5	331	710.2	31	181.4	329	196.2	8032	30231.6
- sat	0	0.0	4667	26301.6	0	0.0	12	170.9	0	0.0	4679	26472.5
- unsat	20	0.3	2654	2841.9	331	710.2	19	10.5	329	196.2	3353	3759.1
iSAT+CAD$_1$	20	0.3	6732	23092.9	331	755.0	31	182.4	328	307.8	7442	24338.4
- sat	0	0.0	4125	20673.8	0	0.0	12	170.4	0	0.0	4137	20844.2
- unsat	20	0.3	2607	2419.0	331	755.0	19	12.0	328	307.8	3305	3494.1
iSAT	20	0.2	2614	285.4	331	719.1	19	10.2	307	2.6	3291	1017.5
- sat	0	0.0	123	2.1	0	0.0	0	0.0	0	0.0	123	2.1
- unsat	20	0.2	2491	283.4	331	719.1	19	10.2	307	2.6	3168	1015.4
SMT+CAD	2	0.2	5358	29494.8	17	4.9	15	1.3	307	581.3	5699	30082.5
- sat	0	0.0	3746	14151.8	0	0.0	11	0.9	0	0.0	3761	14152.7
- unsat	2	0.2	1612	15343.0	17	4.9	4	0.4	307	581.3	1942	15929.8
Z3	11	353.5	8257	593.1	237	2340.7	91	173.5	418	10.0	9014	3470.7
- sat	0	0.0	5350	253.0	7	240.0	61	113.5	0	0.0	5418	606.5
- unsat	11	353.5	2907	340.1	230	2100.6	30	60.0	418	10.0	3596	2864.2
Redlog	3	2.7	7403	28472.2	17	13.6	17	205.9	243	721.9	7683	29416.2
- sat	0	0.0	4969	14023.8	0	0.0	12	42.3	2	143.2	4983	14209.4
- unsat	3	2.7	2434	14448.3	17	13.6	5	163.5	241	578.7	2700	15206.8

8 Outlook

From our results we can conclude that extra input information such as bounds for the variables can aid the CAD search routine and improve its efficiency. Following that line, we want to connect the two procedures even tighter. iSAT is an SMT solver calling the CAD solver for consistency checks, just as a theory solver in the SMT framework. Empirical data from the field of SMT solving shows that a tremendous speed-up can be gained if the respective theory solver supports incremental adding and removing of constraints plus the generation of minimal reasons in case of a conflict and theory deductions in case of no conflict.

Moreover, our results show that, in contrast to a conflict-oriented approach such as in Z3 [12], a solution-oriented implementation of the CAD method as in SMT+CAD is widely inefficient on the SMT benchmarks. However, its adaption and combination with an incomplete ICP solver in a prototypical implementation is already almost at eye level with highly developed SMT solvers such as Z3. We expect that a bound-using adaption of a complete NRA solving strategy not only composed of the CAD method would be even more beneficial.

Our CAD implementation uses arbitrary-precision arithmetic for computing with interval bounds. In contrast, iSAT is mainly implemented based on fast floating-point arithmetic. It is possible to use validated floating-point computations in the CAD method, too. Thus, we could improve the efficiency of our CAD solver and reduce the overhead of the CAD solver calls within iSAT.

Acknowledgments. We appreciate the very interesting and detailed comments by the anonymous reviewers.

References

1. Anai, H., Yokoyama, K.: Cylindrical algebraic decomposition via numerical computation with validated symbolic reconstruction. In: Algorithmic Algebra and Logic, pp. 25–30 (2005)
2. Basu, S., Pollack, R., Roy, M.: Algorithms in Real Algebraic Geometry. Springer (2010)
3. Benhamou, F., Granvilliers, L.: Continuous and Interval Constraints. In: Handbook of Constraint Programming, pp. 571–603. Foundations of Artificial Intelligence (2006)
4. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
5. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
6. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975)
7. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM 5, 394–397 (1962)
8. Eggers, A., Kruglov, E., Kupferschmid, S., Scheibler, K., Teige, T., Weidenbach, C.: Superposition modulo non-linear arithmetic. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 119–134. Springer, Heidelberg (2011)
9. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. Journal on Satisfiability, Boolean Modeling, and Computation 1(3-4), 209–236 (2007)
10. Hong, H.: An improvement of the projection operator in cylindrical algebraic decomposition. In: ISSAC 1990, pp. 261–264. ACM (1990)
11. Iwane, H., Yanami, H., Anai, H.: An effective implementation of a symbolic-numeric cylindrical algebraic decomposition for optimization problems. In: Proceedings of the 2011 International Workshop on Symbolic-Numeric Computation, pp. 168–177. ACM (2012)
12. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 339–354. Springer, Heidelberg (2012)
13. Mishra, B.: Algorithmic Algebra. Texts and Monographs in Computer Science. Springer (1993)
14. Ratschan, S.: Approximate quantified constraint solving by cylindrical box decomposition. Reliable Computing 8(1), 21–42 (2002)
15. Silva, J.P.M., Sakallah, K.A.: Grasp - a new search algorithm for satisfiability. In: ICCAD, pp. 220–227 (1996)
16. Strzebonski, A.W.: Cylindrical algebraic decomposition using validated numerics. Journal of Symbolic Computation 41(9), 1021–1038 (2006)
17. Tseitin, G.S.: On the complexity of derivations in propositional calculus. In: Slisenko, A. (ed.) Studies in q(1968)
18. Weispfenning, V.: The complexity of linear problems in fields. Journal of Symbolic Computation 5(1-2), 3–27 (1988)

dReal: An SMT Solver for Nonlinear Theories over the Reals*

Sicun Gao, Soonho Kong, and Edmund M. Clarke

Carnegie Mellon University, Pittsburgh, PA 15213

Abstract. We describe the open-source tool dReal, an SMT solver for nonlinear formulas over the reals. The tool can handle various nonlinear real functions such as polynomials, trigonometric functions, exponential functions, etc. dReal implements the framework of δ-complete decision procedures: It returns either unsat or δ-sat on input formulas, where δ is a numerical error bound specified by the user. dReal also produces certificates of correctness for both δ-sat (a solution) and unsat answers (a proof of unsatisfiability).

1 Introduction

SMT formulas over the real numbers can encode a wide range of problems in theorem proving and formal verification. Such formulas are very hard to solve when nonlinear functions are involved. Our recent work on δ-complete decision procedures provided a new framework for this problem [10,11]. We say a decision procedure is *δ-complete* for a set S of SMT formulas, where δ is a positive rational number, if for any φ from S, the procedure returns one of the following:

- unsat: φ is unsatisfiable.
- δ-sat: φ^δ is satisfiable.

Here, φ^δ is a syntactic variant of φ that encodes a notion of numerical perturbation on logic formulas [10]. With such relaxation, δ-complete decision procedures can fully exploit the power of scalable numerical algorithms to solve nonlinear problems, and at the same time provide suitable correctness guarantees for many correctness-critical problems. dReal implements this framework. It solves SMT problems over the reals with nonlinear functions, such as polynomials, sine, exponentiation, logarithm, etc. The tool is open-source[1], built on opensmt [5] for the high-level DPLL(T) framework, and realpaver [14] for the Interval Constraint Propagation algorithm. It returns unsat or δ-sat on input formulas, and the user can obtain certificates (proof of unsatisfiability or solution) for the answers.

In this paper we describe the usage, design, and some results of the tool.

* This research was sponsored by the National Science Foundation grants no. DMS1068829, no. CNS0926181 and no. CNS0931985, the GSRC under contract no. 1041377, the Semiconductor Research Corporation under contract no. 2005TJ1366, and the Office of Naval Research under award no. N000141010188.

[1] dReal is available at `http://dreal.cs.cmu.edu`.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 208–214, 2013.

Related Work. SMT solving for nonlinear formulas over the reals has gained much attention in recent years and many tools are now available. The symbolic approaches include Cylindrical Decomposition [6], with significant recent improvement [19,16], and Gröbner bases [20]. A drawback of symbolic algorithms is that it is restricted to arithmetic, namely polynomial constraints, with the exception of [1]. On the other hand, many practical solvers incorporate scalable numerical computations. Examples of numerical algorithms that have been exploited include optimization algorithms [4,18], interval-based algorithms [8,7,12], Bernstein polynomials [17], and linearization algorithms [9]. All solvers show promising results on various nonlinear benchmarks. Our goal is to provide an open-source platform for the rigorous combination of numerical and symbolic algorithms under the framework of δ-complete decision procedures [10].

2 Usage

2.1 Input Format

We accept formulas in the standard SMT-LIB 2.0 format [2] with extensions. In addition to nonlinear arithmetic (polynomials), we allow transcendental functions such as $\sin, \tan, \arcsin$, arctan, exp, log, pow, sinh. More nonlinear functions (for instance, solution of differential equations) can be added when needed, by providing the corresponding numerical evaluation algorithms. Floating-point numbers are allowed as constants in the formula.

Bound information on variables can be declared using a list of simple atomic formulas. For instance "`(assert (< 0 x))`", which sets $x \in (0, +\infty)$ at parsing time. Also, the user can set the precision by writing "`(set-info :precision 0.0001)`." The default precision is 10^{-3}, and can be set through command line.

Example 2.1. The following is an example input file. It is taken from the Flyspeck project [15]. (Filename `flyspeck/172.smt2`. Flyspeck `ID (6096597438b)`)

```
(set-logic QF_NRA)
(set-info :precision 0.001)
(declare-fun x () Real)
(assert (<= 3.0 x))
(assert (<= x 64.0))
(assert (not (> (- (* 2.0 3.14159265) (* 2.0 (* x (arcsin (* (cos
0.797) (sin (/ 3.14159265 x))))))) (+ (- 0.591 (* 0.0331 x))
(+ (* 0.506 (/ (- 1.26 1.0) (- 1.26 1.0))) 1.0)))))
(check-sat)
(exit)
```

2.2 Command Line Options

After building, `dReal` can be simply used through:

```
dReal [--verbose] [--proof] [--precision <double>] <filename>
```

The default output is unsat or delta-sat. When the flags are enabled, the following output will be provided.

- If `--verbose` is set, then the solver will output the detailed decision traces along with the solving process.
- If `--proof` is set, the solver produces an addition file "`filename.proof`" upon termination, and provides the following information.
  - If the answer is delta-sat, then `filename.proof` contains a witnessing solution, plugged into a δ-perturbation of the original formula, such that the correctness can be easily checked externally.
  - If the answer is unsat, then `filename.proof` contains a trace of the solving steps, which can be verified as a proof tree that establishes the unsatisfiability of the formula.
- The `--precision` flag gives the option of overwriting the default precision, and the one set in the benchmark.

When the `--proof` flag is set, the solver produces a file that certifies the answer. In the `delta-sat` case, the solution is plugged in the formula, and its correctness can be checked externally. For the `unsat` cases, we provide a proof checker that verifies the proof. It can be used with the following command:

```
proofcheck [--timeout <int>] <filename>
```

The proof checker will create a new folder called `filename.extra`, which contains auxiliary files needed. It is possible for the proof checking procedure to produce a large number of new files, so setting a timeout is important. By default, the timeout is `30min`. The proof checker will return either "proof verified" or "timeout".

Example 2.2. With default parameters, dReal solves the formula in Example 2.1 in `10ms`, returning unsat, on a machine with a 32-core 2.3GHz AMD Opteron Processor and 94GB of RAM. We then run proofcheck on the same machine. The proof checker returns "proof verified" in 10.08s, after making 8 branching steps and checking 77 axioms.

3 Design

3.1 The δ-Decision Problem

The standard decision problem is undecidable for SMT formulas over the reals with trigonometric functions. Instead, we proposed to focus on the so-called δ-decision problem, which relaxes the standard decision problem. Let δ be any positive rational number. On a given SMT formula φ, we ask for one of the following answers:

- unsat: φ is unsatisfiable.
- δ-sat: φ^δ is satisfiable.

When the two cases overlap, either answer can be returned. Here, φ^δ is called the δ-perturbation (or δ-weakening) of φ, which is formally defined as follows.

Definition 3.1 (δ-Weakening [10]). *Let $\delta \in \mathbb{Q}^+ \cup \{0\}$ be a constant and φ be a Σ_1-sentence in a standard form $\varphi := \exists^{\boldsymbol{I}} \boldsymbol{x}\ (\bigwedge_{i=1}^{m}(\bigvee_{j=1}^{k_i} f_{ij}(\boldsymbol{x}) = 0))$. The δ-weakening of φ defined as: $\varphi^\delta := \exists^{\boldsymbol{I}} \boldsymbol{x}\ (\bigwedge_{i=1}^{m}(\bigvee_{j=1}^{k} |f_{ij}(\boldsymbol{x})| \leq \delta))$.*

Solving the δ-decision problem is as useful as the standard one for many problems. For instance, suppose we perform bounded model checking on hybrid systems, and encode safety properties as an SMT formula φ. Then following standard model checking techniques, if we decide that φ is unsat, then the system is indeed "safe" with in some bounds; if we decide that φ is δ-sat, then the system would become "unsafe" under some δ-perturbation on the system. In this way, when δ is reasonably small, we have essentially taken into account the robustness properties of the system, and can justifiably conclude that the system is unsafe in practice.

3.2 DPLL⟨ICP⟩

Interval Constraint Propagation (ICP) [3] is a constraint solving algorithm that finds solutions of real constraints using a "branch-and-prune" method, combining interval arithmetic and constraint propagation. The idea is to use interval extensions of functions to "prune" out sets of points that are not in the solution set, and "branch" on intervals when such pruning can not be done, until a small enough box that may contain a solution is found. In a DPLL(T) framework, ICP can be used as the theory solver that checks the consistency of a set of theory atoms. We use opensmt [5] for the general DPLL(T) framework, and integrate realpaver [14] which performs ICP. We now describe the design of the interface. A high-level structure of the theory solver is shown in Algorithm 1.

Check and Assert. For incomplete checks in the assert function, we use the pruning operator provided in ICP to contract the interval assignments on all the variables, by eliminating the boxes in the domain that do not contain any solutions. At complete checks, we perform both pruning and branching, and look for one interval solution of the system. That is, we prune and branch on the interval assignment of all variables, and stop when either we have obtained an interval vector that is smaller than the preset error bound, or when we have traversed all the possible branching on the interval assignments.

Backtracking and Learning. We maintain a stack of assignments on the variables, which are mappings from variables to unions of intervals. When we reach a conflict, we backtrack to the previous environment in the pushed stack. We also collect all the constraints that have appeared in the pruning process leading to the conflict. We then turn this subset of constraints into a learned clause and add it to the original formula.

Algorithm 1. Theory Solving in DPLL⟨ICP⟩

input : A conjunction of theory atoms, seen as constraints, $c_1(x_1, ..., x_n), ..., c_m(x_1, ..., x_n)$, the initial interval bounds on all variables $B^0 = I_1^0 \times \cdots \times I_n^0$, box stack $S = \emptyset$, and precision $\delta \in \mathbb{Q}^+$.
output: δ-sat, or unsat with learned conflict clauses.

```
1  S.push(B_0);
2  while S ≠ ∅ do
3      B ← S.pop() ;
4      while ∃1 ≤ i ≤ m, B ≠ Prune(B, c_i) do
5          //Pruning without branching, used as the assert() function.
           B ← Prune(B, c_i);
6      end
7      //The ε below is computed from δ and the Lipschitz constants of
       functions beforehand.
8      if B ≠ ∅ then
9          if ∃1 ≤ i ≤ n, |I_i| ≥ ε then
10             {B_1, B_2} ← Branch(B, i); //Splitting on the intervals
               S.push({B_1, B_2});
11         else
12             return δ-sat;  //Complete check() is successful.
13         end
14     end
15 end
16 return unsat;
```

Witness for δ-Satisfiabillity. When the answer is δ-sat on $\varphi(\boldsymbol{x})$, we provide a solution $\boldsymbol{a} \in \mathbb{R}^n$, such that $\varphi^\delta(\boldsymbol{a})$ is a ground formula that can be easily checked to be true. It is important to note that the solution witnesses δ-satisfiability, instead of standard satisfiability of the original formula. While the latter problem is undecidable, *any* point in the interval assignment returned by ICP can witness the satisfiability of φ^δ when the intervals are smaller than an appropriate error bound.

Proofs of Unsatisfiability. When the answer is unsat, we produce a proof tree that can be verified to establish the validity of the negation of the formula, i.e., $\forall \boldsymbol{x} \neg \varphi(\boldsymbol{x})$. We devised a simple first-order natural deduction system, and transform the computation trace of the solving process into a proof tree. We then use interval arithmetic and simple rules to check the correctness of the proof tree. The proof check procedure recursively divide the problem into subproblems with smaller domains. More details can be found in [13].

4 Results

Besides solving the standard benchmarks [16] (data shown on the tool website), we managed to solve many challenging nonlinear benchmarks from the Flyspeck

project [15] for the formal proof of the Kepler conjecture. The following is a typical formula:

$$\forall \boldsymbol{x} \in [2, 2.51]^6.\ \Big(-\frac{\pi - 4\arctan\frac{\sqrt{2}}{5}}{12\sqrt{2}}\sqrt{\Delta(\boldsymbol{x})} + \frac{2}{3}\sum_{i=0}^{3}\arctan\frac{\sqrt{\Delta(\boldsymbol{x})}}{a_i(\boldsymbol{x})} \leq -\frac{\pi}{3} + 4\arctan\frac{\sqrt{2}}{5}\Big)$$

where $a_i(\boldsymbol{x})$ are quadratic and $\Delta(\boldsymbol{x})$ is the determinant of a nonlinear matrix. We solved 828 out of the 916 formulas (returning unsat) with a timeout of 5 minutes and $\delta = 10^{-3}$, without domain-specific heuristics. The proof traces of these formulas can be large. In Table 1, we list some of the representative benchmarks to show scalability. Complete tables are on the tool page.

Table 1. Experimental results. #OP = Number of nonlinear operators in the problem, TIME_S = Solving time in seconds, TO = Timeout (30min), PC = Proof Checked, #PA = Number of proved axioms, #SP = Number of subproblems generated by proof checking, TIME_PC = Proof-checking time in seconds, #D = Number of iteration depth required in proof checking.

Problem#	#OP	Time_S	Result	Trace Size	PC	#PA	#SP	Time_PC	#D
506	49	0:00.01	UNSAT	519	Y	3,108	3,107	190.200	9
504	48	0:00.01	UNSAT	507	Y	2,322	2,321	172.250	9
746	2,729	0:00.22	UNSAT	20,402	Y	134	135	156.940	99
785	81	0:00.79	UNSAT	2,530,262	Y	1,968	1,454	100.620	5
505	48	0:00.01	UNSAT	477	Y	1,390	1,389	84.030	9
814	96	0:00.50	UNSAT	1,349,482	Y	885	638	79.010	5
783	832	0:00.06	UNSAT	6,386	Y	211	210	57.890	9
815	96	0:00.48	UNSAT	1,394,542	Y	912	688	45.620	5
760	2,792	0:00.22	UNSAT	20,991	Y	71	70	34.470	9
816	97	0:00.15	UNSAT	423,074	Y	335	254	30.310	5
260	90	0:45.10	UNSAT	306,508,373	N	—	—	—	–
884	94	0:25.75	UNSAT	181,766,839	N	—	—	—	–
461	36	0:25.20	UNSAT	133,865,608	N	—	—	—	–
871	80,230	0:16.38	UNSAT	610,809	N	—	—	—	–
525	43	4:38.01	δ-SAT	—	–	—	—	—	–

References

1. Akbarpour, B., Paulson, L.C.: Metitarski: An automatic prover for the elementary functions. In: AISC/MKM/Calculemus, pp. 217–231 (2008)
2. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, Edinburgh, UK (2010)

3. Benhamou, F., Granvilliers, L.: Continuous and interval constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, ch. 16. Elsevier (2006)
4. Borralleras, C., Lucas, S., Navarro-Marset, R., Rodríguez-Carbonell, E., Rubio, A.: Solving non-linear polynomial arithmetic via SAT modulo linear arithmetic. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 294–305. Springer, Heidelberg (2009)
5. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The openSMT solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
6. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Automata Theory and Formal Languages, pp. 134–183 (1975)
7. Eggers, A., Fränzle, M., Herde, C.: SAT modulo ODE: A direct SAT approach to hybrid systems. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 171–185. Springer, Heidelberg (2008)
8. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. JSAT 1(3-4), 209–236 (2007)
9. Ganai, M.K., Ivančić, F.: Efficient decision procedure for non-linear arithmetic constraints using cordic. In: Formal Methods in Computer Aided Design (FMCAD) (2009)
10. Gao, S., Avigad, J., Clarke, E.M.: Delta-complete decision procedures for satisfiability over the reals. In: IJCAR, pp. 286–300 (2012)
11. Gao, S., Avigad, J., Clarke, E.M.: Delta-decidability over the reals. In: LICS, pp. 305–314 (2012)
12. Gao, S., Ganai, M., Ivancic, F., Gupta, A., Sankaranarayanan, S., Clarke, E.: Integrating ICP and LRA solvers for deciding nonlinear real arithmetic. In: FMCAD (2010)
13. Gao, S., Kong, S., Wang, M., Clarke, E.: Extracting proofs from a numerically-driven decision procedure, CMU SCS Technical Report CMU-CS-13-104 (2013)
14. Granvilliers, L., Benhamou, F.: Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. ACM Trans. Math. Softw. 32(1), 138–156 (2006)
15. Hales, T.C.: Introduction to the flyspeck project. In: Coquand, T., Lombardi, H., Roy, M.-F. (eds.) Mathematics, Algorithms, Proofs, Schloss Dagstuhl, Germany. Dagstuhl Seminar Proceedings, vol. 05021, Internationales Begegnungs- und Forschungszentrum für Informatik, IBFI (2005)
16. Jovanovic, D., de Moura, L.M.: Solving non-linear arithmetic. In: IJCAR, pp. 339–354 (2012)
17. Muñoz, C., Narkawicz, A.: Formalization of a representation of Bernstein polynomials and applications to global optimization. Journal of Automated Reasoning (2012) (accepted for publication)
18. Nuzzo, P., Puggelli, A., Seshia, S.A., Sangiovanni-Vincentelli, A.L.: Calcs: Smt solving for non-linear convex constraints. In: Bloem, R., Sharygina, N. (eds.) FMCAD, pp. 71–79. IEEE (2010)
19. Passmore, G.O., Jackson, P.B.: Combined decision techniques for the existential theory of the reals. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) Calculemus/MKM 2009. LNCS (LNAI), vol. 5625, pp. 122–137. Springer, Heidelberg (2009)
20. Platzer, A., Quesel, J.-D., Rümmer, P.: Real world verification. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 485–501. Springer, Heidelberg (2009)

Solving Difference Constraints over Modular Arithmetic

Graeme Gange, Harald Søndergaard, Peter J. Stuckey, and Peter Schachte

Department of Computing and Information Systems,
The University of Melbourne, Victoria 3010, Australia
{ggange,harald,pjs,schachte}@csse.unimelb.edu.au

Abstract. Difference logic is commonly used in program verification and analysis. In the context of fixed-precision integers, as used in assembly languages for example, the use of classical difference logic is unsound. We study the problem of deciding difference constraints in the context of modular arithmetic and show that it is strongly NP-complete. We discuss the applicability of the Bellman-Ford algorithm and related shortest-distance algorithms to the context of modular arithmetic. We explore two approaches, namely a complete method implemented using SMT technology and an incomplete fixpoint-based method, and the two are experimentally evaluated. The incomplete method performs considerably faster while maintaining acceptable accuracy on a range of instances.

1 Introduction

We consider the problem of adapting classical difference logic over $\mathbb{Z}$ [2] to the congruence class used in modulo-m integer arithmetic, here denoted $\mathbb{Z}_m$. Understanding this class is important for the design of automated reasoning that is concerned with machine arithmetic. Our particular interest in this arises from our work on analysis and verification of low-level code. We wish to improve static analysis techniques for low-level programming languages that use w-bit fixed-precision integers, that is, we are interested in the particular case $m = 2^w$. Much of the literature on program analysis and software verification uses difference logic and similar numeric abstract domains, tacitly assuming *unbounded* integers. It is well known that, in that context, difference logic can be decided in $O(|V||C|)$ deterministic time, for variables V and constraints C. In the context of $\mathbb{Z}_m$, the decision problem becomes strongly NP-complete.

Consider the program fragment shown in Figure 1. Conventional static analysis with difference bound matrices [4] or octagons [11] will derive the bounded difference constraint $0 \leqslant y - x \leqslant 6$ and determine that the branch $y < x$ will never be executed. In a context of fixed-precision integers, owing to possible overflow, this conclusion is clearly wrong. Nevertheless, in some sense the derived invariant is meaningful, as y lies between x and $x + 6$ on the modular integer number circle.

The challenge is to ensure that program analysis "understands" machine operations so as to remain faithful to machine arithmetic. Previous work has mainly

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 215–230, 2013.

```
unsigned int x = ⋆;
unsigned int y = x;
for(int i = 0; i < 6; i ++)
    if(⋆)
        y ++;
if(y < x) ERROR;
```

Fig. 1. Unbounded relational analysis will deem **ERROR** unreachable; however, on systems with fixed-width integers, y may wrap to 0 if x is very large

dealt with non-relational abstract domains, notably *interval* domains [15,16,8]. Simon and King [17] considered adapting convex polyhedra to modular arithmetic by computing a convex approximation relative to a fixed wrapping point. Other approaches consider instead systems of equations [13] and disequations [9] under modular arithmetic. $\mathbf{SMT}(\mathcal{BV})$ [10] problems involve a variety of constraints over $\mathbb{Z}_{2^w}$; solvers for these problems typically convert the arithmetic operations to SAT. These techniques are complete, but may be too slow to be viable for certain applications, such as invariant synthesis.

One critical issue with classical abstract domains such as interval domains, octagons, and so on, is that they rely on having a linear ordering, $\leqslant$, on the set of integers. The only way to capture a non-trivial concept of ordering on $\mathbb{Z}_m$ is to discuss order only with respect to some reference point. For example, we may decide that $x \leqslant y$ means, loosely, that "starting from 0 and moving clockwise on the number circle, x is encountered no later than y"—a natural reading when unsigned integer representation is used. Or, we may decide that it means "starting from $-m/2$, x is met no later than y"—when signed representation is used. To complicate matters, many low-level languages, such as LLVM and assembly languages, fail to provide signedness information, relying on the fact that arithmetic operations such as addition, subtraction and multiplication are agnostic with respect to signedness. Navas *et al.* [14] point out that analysis of such languages, in order to maintain precision, has to be signedness-agnostic as well, which means superposing signed/unsigned assumptions during analysis.

For the remainder of this paper, we assume all inequalities are unsigned. Signed inequalities $x \leqslant_{\mathrm{s}} y$ can be expressed in terms of unsigned inequalities: $x \leqslant_{\mathrm{s}} y$ iff $x + \frac{m}{2} \leqslant y + \frac{m}{2}$. This does, however, require the introduction of shifted variables $x' = (x + \frac{m}{2}) \bmod m$ and $y' = (y + \frac{m}{2}) \bmod m$.

Classical integer difference constraints provide lower and upper bounds on integer differences $x - y$, and these bounds have consequences for order. For example, for positive k, a constraint $x - y \geqslant k$ allows us to deduce $x \geqslant y$. When we move to $\mathbb{Z}_m$, this link between difference and order is lost. For example, assuming signed arithmetic, $[x \mapsto -2^{w-1}, y \mapsto 2^{w-1} - 1]$ satisfies $x - y \geqslant 0$ but not $x \geqslant y$. Hence an important step towards getting a handle on "wrapped" difference logic is to separate the aspects of proximity and (relative) order.

```
bellman_ford(⟨V, E⟩)
    % Introduce a fresh least element.
    V' = V ∪ {v'}
    E' = E ∪ {⟨v', 0, v_i⟩ | v_i ∈ V}
    % Initialize relations
    D(v') := 0
    for(v_i ∈ V)
        D(v_i) := −∞
    % Progressively expand the set of paths to each node.
    for(k ∈ {1, ..., |V|})
        for((⟨v_i, w, v_j⟩ ∈ E')
            D(v_j) = max(D(v_j), D(v_i) + w)
    % Check for any inconsistencies
    for((⟨v_i, w, v_j⟩ ∈ E')
        if(D(v_i) + w > D(v_j))
            return UNSAT
    return SAT
```

Fig. 2. Pseudo-code for the Bellman-Ford algorithm for checking satisfiability of a set of unbounded difference logic constraints

The following contributions are made in this paper:

- We study the complications that arise when reasoning about difference constraints takes place in the presence of modular arithmetic.
- We offer a simple proof that, in that context, for $m > 2$, decidability of difference constraints is NP-complete.
- We propose a framework for combined reasoning about proximity and order and use this to develop an efficient but incomplete decision procedure.
- We evaluate the resulting method, comparing it to two more traditional SMT-based decision procedures.

In Section 2 we recapitulate the classical case. In Section 3 we discuss "wrapped" difference constraints and develop the different approaches: a complete method based on $\mathbf{SMT}(\mathcal{BV})$ (bit-vector) technology, one using $\mathbf{SMT}(\mathcal{DL})$ (difference logic), and an incomplete fixpoint-based method. Section 4 contains the evaluation and Section 5 concludes.

2 Deciding Difference Logic

The classical method for deciding difference logic[1] is the Bellman-Ford algorithm [2]. For later reference, we show it in Figure 2. It uses a graph

[1] The term "separation logic" is sometimes used [18,19]. To avoid confusion with Reynolds-O'Hearn separation logic, we use the alternative "difference logic".

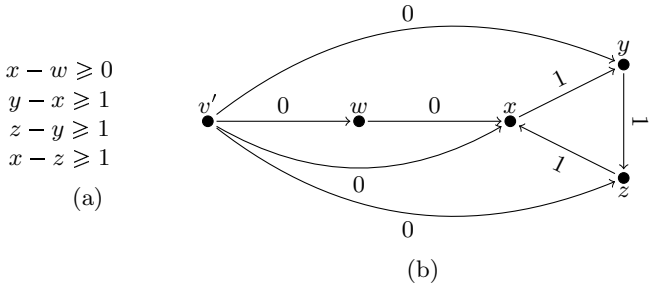


Fig. 3. (a) A set of difference constraints. (b) the corresponding graph representation (v' is a freshly introduced root vertex).

representation of constraints, each variable giving rise to a node, and each difference constraint giving rise to a weighted directed edge. The algorithm relies on these inference rules for difference logic ($\bot$ denotes unsatisfiability):

Inversion:	$\alpha \leqslant y - x \leqslant \beta$ iff $-\beta \leqslant x - y \leqslant -\alpha$
Resolution:	$\dfrac{\alpha_1 \leqslant y - x \leqslant \beta_1 \quad \alpha_2 \leqslant z - y \leqslant \beta_2}{\alpha_1 + \alpha_2 \leqslant z - x \leqslant \beta_1 + \beta_2}$
Tightening:	$\dfrac{\alpha_1 \leqslant y - x \leqslant \beta_1 \quad \alpha_2 \leqslant y - x \leqslant \beta_2}{\mathbf{max}(\alpha_1, \alpha_2) \leqslant y - x \leqslant \mathbf{min}(\beta_1, \beta_2)}$
Contradiction:	$\dfrac{\alpha \leqslant y - x \leqslant \beta, \alpha > \beta}{\bot}$

Example 1. Consider the set of constraints shown in Figure 3(a). The graph corresponding to these constraints is given in Figure 3(b). Note that the constraints are satisfiable if and only if there are no positive-weight cycles in the graph.[2] After computing the longest paths of length up to $|V|$, we have:

$$\{D(v') = 0, D(w) = 0, D(x) = 3, D(y) = 3, D(z) = 3\}$$

However, performing another iteration would still increase the path lengths, since, for example, $D(z) + 1 > D(x)$. This indicates the presence of a positive-weight cycle. □

3 Wrapped Difference Constraints

In this section, we consider systems of constraints of the form $\alpha \in y - x \in \beta$ (which we sometimes write it as $y - x \in [\alpha, \beta]$) under modular arithmetic. Given that

[2] Presentations of difference logic sometimes express the problem in terms of shortest (rather than longest) paths, in which case unsatisfiability corresponds to negative (rather than positive) cycles.

we are concerned with numerical *proximity* as well as ordering, we allow these intervals to cross m; such a *wrapped* interval $[\alpha, \beta]$ is interpreted as follows:[3]

$$\gamma[\alpha,\beta] = \begin{cases} \{d \mid \alpha \leqslant d \wedge d \leqslant \beta\} & \textbf{if } \alpha \leqslant \beta \\ \{d \mid \alpha \leqslant d \wedge d \leqslant m-1\} \cup \{d \mid 0 \leqslant d \wedge d \leqslant \beta\} & \textbf{otherwise} \end{cases}$$

For example, in a modulo-16 context, the wrapped interval $[14, 2]$ denotes the set $\{0, 1, 2, 14, 15\}$. In this modulo-m context, a one-sided constraint (such as $y - x \geqslant 3$) is essentially meaningless; the usual inequalities under $\mathbb{Z}_m$ are implicitly bounded by 0 and $m - 1$. On the other hand, containment of wrapped intervals is easily expressed:

$$[\alpha,\beta] \sqsubseteq [\alpha',\beta'] \text{ iff } \gamma[\alpha,\beta] \subseteq \gamma[\alpha',\beta']$$

We can perform certain operations, similar to those of Section 2, on the number circle. However, of Section 2's inference rules, only inversion remains sound.

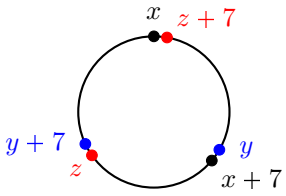
Example 2. Consider, for $k > 1$, this set of constraints:

$$1 \leqslant y - x \leqslant k, \quad 1 \leqslant z - y \leqslant k, \quad 1 \leqslant x - z \leqslant k \tag{1}$$

Resolving the first two constraints, we get $2 \leqslant z - x \leqslant 2k$, or, equivalently, $-2k \leqslant x - z \leqslant -2$. Under standard difference logic, (1) would be unsatisfiable:

$$\dfrac{\dfrac{1 \leqslant x - z \leqslant k \qquad -2k \leqslant x - z \leqslant -2}{1 \leqslant x - z \leqslant -2}}{\bot}$$

However, as illustrated here, in the modular-arithmetic case, this set of constraints is satisfiable if k is sufficiently large. For example, the constraints are satisfiable in $\mathbb{Z}_{16}$, for $k = 7$ (take, say, $x = 1, y = 6, z = 11$). □



The n constraints

$$\alpha_1 \leqslant x_2 - x_1 \leqslant \beta_1, \ldots, \alpha_{n-1} \leqslant x_n - x_{n-1} \leqslant \beta_{n-1}, \alpha_n \leqslant x_1 - x_n \leqslant \beta_n$$

induce the constraint $\alpha \leqslant 0 \leqslant \beta$, where $\alpha = \alpha_1 + \cdots + \alpha_n$ and $\beta = \beta_1 + \cdots + \beta_n$. This latter constraint is unsatisfiable exactly when 0 falls outside $[\alpha, \beta]$, a condition which is reminiscent of the positive-weight cycle condition for conventional difference logic.

In principle a variant of a shortest-path algorithm could be used to detect these inconsistent cycles; however, this requires computing the intersection of wrapped intervals. Consider the two intervals shown in Figure 4(a). The intersection of

[3] The definition overloads the square bracket notation: The function γ takes a possibly *wrapped* interval and expresses its meaning in terms of ordinary intervals.

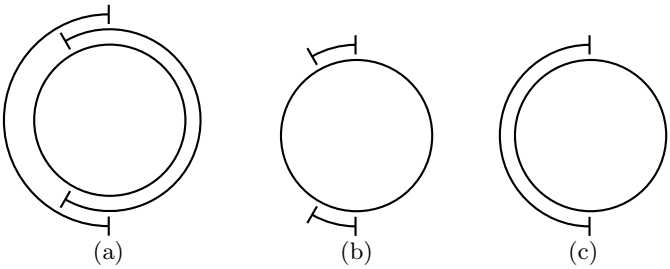


Fig. 4. (a) A pair of intervals on the number circle, (b) the intersection of the two intervals and (c) an optimal over-approximation of the intersection

these two intervals is no longer a single interval. Indeed, the intersection of k wrapped intervals produces up to $min(k, \frac{m}{2})$ disjoint feasible intervals; when combined with resolution, $min(2^{\frac{k}{2}}, \frac{m}{2})$ intervals can be generated. For example, with these constraints:

$$0 \leqslant y - x \leqslant 2,\ 0 \leqslant z - y \leqslant 4,\ 2 \leqslant y - x \leqslant m,\ 4 \leqslant z - y \leqslant m$$

we have $y - x \in \{0, 2\}$ and $z - y \in \{0, 4\}$, which yields $z - x \in \{0, 2, 4, 6\}$. There are four equally good interval approximations of this set.

3.1 Interval Sets

We could represent the feasible relations between two variables exactly by explicitly maintaining the set of (disjoint) feasible intervals. To reason about bounded difference constraints, we require two operations: intersection of interval-sets (denoted $\sqcap$) and pointwise addition of interval-sets (denoted $+$).

$$A \sqcap A' = \bigsqcup_{[\alpha,\beta]\in A} \bigsqcup_{[\alpha',\beta']\in A'} \left\{ \begin{array}{ll} [\alpha, \beta] & \textbf{if } \alpha \in [\alpha', \beta'] \wedge \beta \in [\alpha', \beta'] \\ [\alpha, \beta'] & \textbf{if } \alpha \in [\alpha', \beta'] \wedge \beta' \in [\alpha, \beta] \\ [\alpha', \beta] & \textbf{if } \alpha' \in [\alpha, \beta] \wedge \beta \in [\alpha', \beta'] \\ [\alpha', \beta'] & \textbf{if } \alpha' \in [\alpha, \beta] \wedge \beta' \in [\alpha, \beta] \end{array} \right\}$$

$$A + A' = \bigsqcup_{[\alpha,\beta]\in A} \bigsqcup_{[\alpha',\beta']\in A'} \{[\alpha + \alpha', \beta + \beta']\}$$

The operation $\sqcap$ can be implemented in $O(|A|+|A'|)$ time; $+$ requires $O(|A||A'|)$ time in the worst case. In the case of $+$, the results are normalized by merging overlapping intervals. Note that $+$ and $\sqcap$ are both commutative and associative. Also note that the full interval, $\top$, is a neutral element for $\sqcap$, while $\{[0, 0]\}$ is neutral for $+$. It would be convenient if the resulting structure formed a semiring, as this would allow us to use the algebraic shortest distance framework of Mohri [12]. Unfortunately, while we do have the property

$$(a \sqcap b) + c \sqsubseteq (a + c) \sqcap (b + c)$$

it is *not* the case that $+$ distributes over $\sqcap$. As a counter-example, consider $\mathbb{Z}_{16}$, and take $A = [0, 8]$ and $B = [9, 0]$. We have $A + (A \sqcap B) = A + [0, 0] = A$, while $(A + A) \sqcap (A + B) = \top \sqcap \top = \top$. As we shall see in Section 3.6, this complicates the operation of longest path algorithms.

3.2 Wrapped-Interval Approximation

For the analysis of programs that use $\mathbb{Z}_m$ for a large m (and usually m is 2^{32} or 2^{64}), representing the feasible values precisely is impractical. In this section, we propose the construction of an over-approximation of the set of feasible intervals.

We adopt a "wrapped interval" representation [14], approximating the set of feasible intervals with a single interval. A wrapped interval is any sequence of consecutive numbers on the modulo-m number circle; for example, with $m = 16$, the interval $[8, 0]$ is the set $\{8, 9, \ldots, 15, 0\}$. Given a set of integers modulo m, there may be several minimal approximations in the form of wrapped intervals; for example, the set $\{0, 8\}$ may be approximated by $[0, 8]$ or by $[8, 0]$, two intervals of equal cardinality. To ensure a deterministic choice, we use a total ordering $\preceq$ over wrapped intervals, ordering them primarily by cardinality and then lexicographically (we write $x \ \oplus_m \ y$ for $(x \oplus y) \bmod m$ for binary infix operator $\oplus$):

$$[\alpha, \beta] \preceq [\alpha', \beta'] \textbf{ iff } (\beta -_m \alpha) < (\beta' -_m \alpha') \vee ((\beta -_m \alpha = \beta' -_m \alpha') \wedge \alpha \leqslant \alpha')$$

This allows us to define an over-approximation of the meet which selects the approximation with minimum cardinality, breaking ties by favouring the lexicographically smallest left-bound.

$$[\alpha, \beta] \sqcap_{\mathbb{L}} [\alpha', \beta'] = \begin{cases} \textbf{if} & \alpha \notin [\alpha', \beta'] \wedge \alpha' \notin [\alpha, \beta] & \textbf{then} & \bot \\ \textbf{else if} & [\alpha, \beta] \sqsubseteq [\alpha', \beta'] & \textbf{then} & [\alpha, \beta] \\ \textbf{else if} & [\alpha', \beta'] \sqsubseteq [\alpha, \beta] & \textbf{then} & [\alpha', \beta'] \\ \textbf{else if} & \alpha' \notin [\alpha, \beta] & \textbf{then} & [\alpha, \beta'] \\ \textbf{else if} & \alpha \notin [\alpha', \beta'] & \textbf{then} & [\alpha', \beta] \\ \textbf{else} & \textbf{min}_{\preceq}([\alpha, \beta], [\alpha', \beta']) & & \end{cases}$$

$$[\alpha, \beta] + [\alpha', \beta'] = \begin{cases} \top & \textbf{if } (\beta -_m \alpha) + (\beta' -_m \alpha') \geqslant m - 1 \\ [\alpha +_m \alpha', \beta +_m \beta'] & \textbf{otherwise} \end{cases}$$

Notice that, using this approximation, $\sqcap_{\mathbb{L}}$ lacks several properties provided by typical lattice operations. $\sqcap_{\mathbb{L}}$ is absorptive and commutative but not associative.

Example 3. With $m = 16$, consider $A = [8, 15]$, $B = [12, 9]$, and $C = [0, 10]$. We have $(A \sqcap_{\mathbb{L}} B) \sqcap_{\mathbb{L}} C = [8, 15] \sqcap_{\mathbb{L}} [0, 10] = [8, 10]$. On the other hand, we have $A \sqcap_{\mathbb{L}} (B \sqcap_{\mathbb{L}} C) = [8, 15] \sqcap_{\mathbb{L}} [0, 9] = [8, 9]$. □

Also, $\sqcap_{\mathbb{L}}$ is not monotone (nor decreasing) with respect to the inclusion ordering $\sqsubseteq$ (however, $\sqcap_{\mathbb{L}}$ *is* monotone with respect to the cardinality ordering $\preceq$).

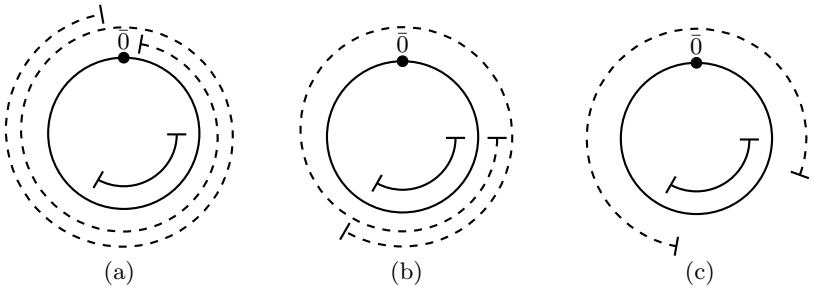


Fig. 5. (a) When the concrete range covers the entire circumference of the number circle, the proximity bounds cannot be reduced further. (b) The concrete bounds can, however, be reduced to the next corresponding proximity bound. (c) If both sets of endpoints are mutually contained, there can be no reduction.

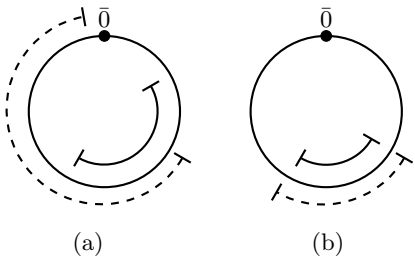


Fig. 6. If the union of the concrete and proximity intervals do not cover the entire number circle, each can be tightened to the region satisfying both

Example 4. Consider the intervals $A = [0, 10]$, $A' = [0, 8]$, $B = [8, 1]$ on $\mathbb{Z}_{16}$. A' is clearly a subset of A, however $A \sqcap_{\mathbb{L}} B$ is incomparable with $A' \sqcap_{\mathbb{L}} B$. Namely, $A \sqcap_{\mathbb{L}} B = [8, 1]$, while $A' \sqcap_{\mathbb{L}} B = [0, 8]$. So $\sqcap_{\mathbb{L}}$ fails to be monotone. □

3.3 Combining Wrapped Difference with Relative Order

The wrapped interval constraints discussed so far express proximity only. They cannot express constraints such as $x < y$. This can be fixed, however, by allowing "concrete" interval information. Thus we combine proximity and range constraints in pairs $\langle [\alpha, \beta], [d, D] \rangle$ with the semantics:

$$\gamma\langle [\alpha, \beta], [d, D] \rangle = \{(x, y) \in \mathbb{Z}_m^2 \mid y -_m x \in [\alpha, \beta] \wedge y - x \in [d, D]\}$$

Note that, with $x, y \in [0, m-1]$, the value of $y-x$ can be anywhere between $-m+1$ and $m-1$ ($2m-1$ possible values). Hence $\top = \langle [0, m-1], [-m+1, m-1] \rangle$. Figure 5(a) depicts an interval pair (assuming $m = 16$, the pair is $\langle [4, 9], [-15, 15] \rangle$), the first interval shown by a solid arc, the second by a dashed arc.

We normalise interval pairs by propagating information from each component to the other. Let d_m be d projected onto the range $[0, m-1]$. We can use this to determine how far the lower (respectively upper) bound of the concrete range must be adjusted to reach the corresponding proximity bound. This difference is then mapped back onto the concrete range. Note the use of $[d, D]_m$, the projection of the interval onto $[0, m-1]$, defined as $[d, D]_m = [0, m-1]$ if $D - d \geqslant m$, and $[d, D]_m = [d_m, D_m]$, otherwise.

$$\begin{aligned}
&\textbf{norm}\ \langle [\alpha, \beta], [d, D] \rangle = \langle [\alpha', \beta'], [d', D'] \rangle \quad \textbf{where} \\
&\qquad d' \quad = \begin{cases} d & \textbf{if } d_m \in [\alpha, \beta] \\ d + (\alpha -_m d_m) & \textbf{otherwise} \end{cases} \\
&\qquad D' \quad = \begin{cases} D & \textbf{if } D_m \in [\alpha, \beta] \\ D - (D_m -_m \beta) & \textbf{otherwise} \end{cases} \\
&\qquad [\alpha', \beta'] = \begin{cases} [\alpha, \beta] & \textbf{if } \alpha \in [d, D]_m \wedge d_m \in [\alpha, \beta] \\ [\alpha, \beta] \sqcap [d, D]_m & \textbf{otherwise} \end{cases}
\end{aligned}$$

Details of the definition are justified by considering the cases shown in Figures 5 and 6. The following theorem says that **norm** establishes the tightest possible consistent bounds.

Theorem 1. Let $\langle A', C' \rangle = \textbf{norm}\ \langle A, C \rangle$. For all A'', C'', if $\gamma\langle A'', C'' \rangle = \gamma\langle A, C \rangle$ then $A' \sqsubseteq A'' \wedge C' \sqsubseteq C''$.

Proof. Consider a pair $\langle [\alpha', \beta'], [d', D'] \rangle = \textbf{norm}\ \langle [\alpha, \beta], [d, D] \rangle$. In the case that $\alpha \in [d_m, D_m]$ and $d_m \in [\alpha, \beta]$, as illustrated in Figure 5(c), neither bound can be tightened. If $[\alpha, \beta]$ and $[d, D]_m$ do not intersect, we have $d + (\alpha -_m d_m) > D$, and the result correctly represents $\bot$.

This leaves the case where there is some overlap between $[\alpha, \beta]$ and $[d, D]_m$, but the intervals do not cross at both ends. In that case, $[\alpha, \beta] \sqcap [d, D]_m$ is the largest interval consistent with both $[\alpha, \beta]$ and $[d, D]$. If $d_m \notin [\alpha, \beta]$, we must adjust d to the next point on the number circle consistent with $[\alpha, \beta]$ – that is, α. The minimum distance d must be shifted is $\alpha -_m d_m$, in which case $d'_m = \alpha$. By similar reasoning, if $D_m \notin [\alpha, \beta]$, we must reduce D by $D_m -_m \beta$, giving $D'_m = \beta$. Both d'_m and D'_m are in $[\alpha', \beta']$. Assume there were some element of $c \in [\alpha', \beta']$ such that $c \notin [d', D']_m$. As $c \in [\alpha', \beta']$, we have $c \in [\alpha, \beta] \wedge c \in [d, D]$. Then c is either in the interval $[d, d')$, or $(D', D]$. However, this cannot be the case, as there are no elements of $[\alpha, \beta]$ in either interval. Therefore, all elements of $[\alpha', \beta']$ must be consistent with $[d', D']$.

We conclude that **norm** computes the tightest intervals that preserve the semantics of the input pair. □

In the case of the complete interval set representation, $[\alpha', \beta']$ can be computed with a standard join; the additional case is to avoid losing information when the intervals overlap in two places (analogous to the case in Figure 4(a)). Using normalisation, we can define the necessary operators on the combined domain:

$$\langle [\alpha_x, \beta_x], [d_x, D_x] \rangle \sqcap \langle [\alpha_y, \beta_y], [d_y, D_y] \rangle = \\ \mathbf{norm} \langle [\alpha_x, \beta_x] \sqcap [\alpha_y, \beta_y], [\mathbf{max}(d_x, d_y), \mathbf{min}(D_x, D_y)] \rangle$$

$$\langle [\alpha_x, \beta_x], [d_x, D_x] \rangle + \langle [\alpha_y, \beta_y], [d_y, D_y] \rangle = \\ \mathbf{norm} \langle [\alpha_x, \beta_x] + [\alpha_y, \beta_y], [d_x + d_y, D_x + D_y] \rangle$$

3.4 NP-Completeness of Wrapped Difference Constraints

As already observed by Bjørner *et al.* [1], wrapped difference constraints are NP-complete. In this section we give a simpler proof, using, as do Bjørner *et al.*, reduction from graph 3-colourability. We strengthen the result [1] by showing NP-completeness for all cases $m > 2$. For $m = 2$, the problem can be solved in polynomial time in the same manner as 2-colouring.

First, given an assignment to variables $\{v_1, \ldots, v_n\}$, we can check, in polynomial time, whether each difference constraint is satisfied; so wrapped difference logic is in NP. It remains to show that the problem is NP-hard.

Assume $m > 2$; consider a 3-COLOURABILITY instance $G = \langle \{v_1, \ldots, v_n\}, E \rangle$. We construct a system of $|E| + n$ difference constraints in $\mathbb{Z}_m$ over variables $\{x, x_1, \ldots, x_n\}$:

- For each vertex v_i, introduce the constraint $x_i - x \in [0, 2]$ (for $m = 3$ this is a vacuous constraint, so it can be omitted).
- For each edge $(v_i, v_j) \in E$, introduce the constraint $x_i - x_j \in [1, m - 1]$.

The system of constraints can be generated in linear time. We claim that it is satisfiable iff G is 3-colourable.

Assume the set of constraints can be satisfied and let ν be a satisfying valuation. For each x_i, we have $\nu(x_i) \in \{\nu(x), \nu(x) +_m 1, \nu(x) +_m 2\}$, owing to the constraint $x_i - x \in [0, 2]$. Taking $\nu(x)$, $\nu(x) +_m 1$, and $\nu(x) +_m 2$ as three "colours", we choose the colour $\nu(x_i)$ for node v_i. This gives a 3-colouring of G, because, for adjacent vertices v_i and v_j, the colours $\nu(x_i)$ and $\nu(x_j)$ must be different, owing to the constraint $x_i - x_j \in [1, m - 1]$.

Conversely, assume that G is 3-colourable. Call the three colours used in the colouring 0, 1, and 2. We claim that the valuation ν which maps x to 0 and x_i to the colour of v_i satisfies the generated constraints. The constraints of form $x_i - x \in [0, 2]$ are satisfied by construction. The constraints of form $x_i - x_j \in [1, m - 1]$ are similarly satisfied, as the "difference" between the "colours" of v_i and v_j are precluded from being 0.

It follows that wrapped difference logic is NP-complete. Note that the usual directionality of edges in the graph generated by difference constraints is irrelevant here, since in modulo m arithmetic, $x - y \in [1, m - 1]$ and $y - x \in [1, m - 1]$ (and $x \neq y$) are equivalent. Also note that the reduction does not synthesize m from a 3-COLOURABILITY instance. Rather, m is a fixed constant in the transformation. As 3-COLOURABILITY is strongly NP-complete, and the transformation is pseudo-polynomial [7], wrapped difference logic is also strongly NP-complete.

3.5 SMT Encodings: Two Complete Decision Procedures

A common approach for solving problems over $\mathbb{Z}_{2^w}$ is *satisfiability modulo bit-vectors* (**SMT**($\mathcal{BV}$)) [10]. In an **SMT**($\mathcal{BV}$) solver, each w-bit word x is typically translated into a vector v_x of w Boolean variables. Operations on $\mathbb{Z}_{2^w}$ are encoded using Boolean formulae to simulate the corresponding hardware circuit.

We can readily couch wrapped difference constraints in terms of **SMT**($\mathcal{BV}$). Letting $\texttt{-}_{\texttt{bv}}$ denote w-bit bit-vector subtraction, encode each constraint directly:

For a constraint $x \leqslant y$:	$v_x \leqslant_{\mathrm{u}} v_y$
For a constraint $y - x \in [i, j]$:	$(v_y \texttt{-}_{\texttt{bv}} v_x) \texttt{-}_{\texttt{bv}} i \leqslant_{\mathrm{u}} j \texttt{-}_{\texttt{bv}} i$

SMT($\mathcal{BV}$) solvers typically use complete methods for solving bit-vector constraints. These, then, provide a complete decision procedure for wrapped difference constraints.

An alternative way is to use *satisfiability modulo difference logic* (**SMT**($\mathcal{DL}$)). Each variable is constrained to the interval $[\mathit{zero}, \mathit{zero} + m - 1]$. We encode the concretization of a wrapped interval $[i, j]$ as a disjunction of concrete difference constraints, using similar reasoning to that illustrated in Figure 5:

For a variable x:	$0 \leqslant v_x - \mathit{zero} \leqslant m - 1$
For a constraint $x \leqslant y$:	$v_x \leqslant_{\mathrm{u}} v_y$
For a constraint $y - x \in [i, j]$:	$\begin{cases} \left(\begin{array}{l} -m + 1 \leqslant v_y - v_x \leqslant -m + j \\ \vee\ -m + i \leqslant v_y - v_x \leqslant j \\ \vee\ i \leqslant v_y - v_x \leqslant m - 1 \end{array}\right) & \textbf{if } j_m < i_m \\ \left(\begin{array}{l} -m + i \leqslant v_y - v_x \leqslant -m + j \\ \vee\ i \leqslant v_y - v_x \leqslant j \end{array}\right) & \textbf{otherwise} \end{cases}$

3.6 An Incomplete Decision Procedure

Ideally, we would like an efficient, sound and complete decision procedure. Given that wrapped difference constraints are NP-complete, it seems highly unlikely that such a procedure exists. The SMT approaches are sound and complete, but can exhibit exponential running time. For use in an abstract interpretation framework, we require the analysis to be efficient, and we can afford to sacrifice completeness (but not soundness). We must therefore develop a sound over-approximation which maintains reasonable accuracy without excessive cost.

Given the similarities between wrapped difference constraints and classical difference logic, it seems likely that variants of shortest-path algorithms would provide suitable heuristics. Indeed, the problem of detecting a set of inconsistent wrapped difference constraints is very similar to the algebraic shortest distance framework of Mohri [12]. As observed in Section 3.1, our $\sqcap$ and $+$ operators lack some critical semiring properties; however, the structure of the problem remains the same. It is worth noting that all edges have an inverse; for any

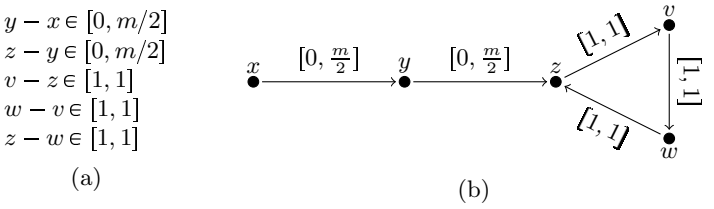


Fig. 7. The cycle $z \to w \to v \to z$ is inconsistent. However, this information is lost when applying Bellman-Ford from the root x.

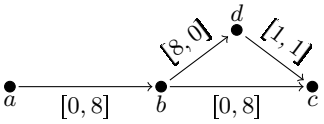


Fig. 8. After running the Floyd-Warshall algorithm (with variables ordered lexicographically), we have $c - a \in \top$, rather than the tightest bound of $[0, 9]$

edge $y - x \in [\alpha, \beta]$, there is a corresponding edge $x - y \in [m - \beta, m - \alpha]$. As the inverse is easily computed, there is no need to store both edges explicitly. Similarly, we do not need to compute $(z-y)+(y-x)$ if we have already computed $(x - y) + (y - z)$.

In principle, the Bellman-Ford algorithm [2] provides a suitable sound over-approximation. Unfortunately, in the context of modular arithmetic, it quickly loses information about infeasible paths.

Example 5. Consider the set of constraints in Figure 7. The cycle $z \to w \to v \to z$ has value $[3, 3]$, which is an inconsistent self-loop (assuming $m > 3$). However, applying Bellman-Ford from root node x, we quickly determine that $z - x \in \top$. Then, as $\top + [1, 1] = \top$, we derive the same relation for $v - x$ and $w - x$. We cannot then deduce the existence of an inconsistent cycle. □

This example suggests that a single-source approach is unlikely to work. An alternative approach is to use an all-pairs shortest path algorithm to derive the strongest relation between each pair of variables. The obvious algorithm to use is the Floyd-Warshall algorithm [5]. However, as mentioned in Section 3.1, + does not distribute over $\sqcap$; as a result, a direct application of the Floyd-Warshall algorithm is not guaranteed to reach a fixpoint with respect to every pair of variables.

Example 6. Consider a problem in $\mathbb{Z}_{16}$, with the constraint graph given in Figure 8. The algorithm computes the longest paths via first a then b (neither tightening any constraints). Since $d - c \in [15, 15]$ and $c - b \in [0, 8]$, paths via c tighten $d - b$ to $[8, 0] \sqcap [15, 7] = [15, 0]$ (note that $c - b$ is still $[0, 8]$, and $d - a$ is still $\top$). Once we compute paths via d, we tighten $c - b$ to $[0, 1]$. After the algorithm has finished, $c - a$ has still not yet been tightened to the correct value of $[0, 9]$. □

```
wrapdiff_fixpoint(⟨V, E⟩)
    Q := ∅
    % Initialize relations
    R := {(v_i, v_j) ↦ ⊤ | v_i, v_j ∈ V}
    Adj := {v_i ↦ ∅ | v_i ∈ V}
    for(⟨v_j − v_i ∈ r⟩ ∈ E)
        update_rel(v_i, v_j, r ⊓ R(v_i, v_j))
    while(¬Q.empty())
        (v_i, v_j) := Q.pop()
        for(v_k ∈ Adj(v_j)\{v_i})
            r_ik := R(v_i, v_k) ⊓ (R(v_i, v_j) + R(v_j, v_k))
            if(r_ik = ⊥) return UNSAT
            update_rel(v_i, v_k, r_ik)
        for(v_k ∈ Adj(v_i)\{v_j})
            r_kj := R(v_k, v_j) ⊓ (R(v_k, v_i) + R(v_i, v_j))
            if(r_kj = ⊥) return UNSAT
            update_rel(v_k, v_j, r_kj)
    % If we reach a fixpoint without unsatisfiability,
    % assume satisfiability
    return SAT

update_rel(v_i, v_j, r_ij)
    if(r_ij ≠ R(v_i, v_j))
        R(v_i, v_j) := r_ij
        Q.insert((v_i, v_j))
        Adj(v_i) := Adj(v_i) ∪ {v_j}
        Adj(v_j) := Adj(v_j) ∪ {v_i}
```

Fig. 9. Computation of a fixpoint of a set of difference constraints

We could modify the Floyd-Warshall algorithm to continue iterating until a fixpoint is reached. However, this performs a great deal of redundant work; particularly given that most such constraint systems are quite sparse, so many edges are $\top$. Instead, we use a worklist-based algorithm to compute the fixpoint directly. We maintain a queue of (v_i, v_j) pairs which have changed, and update any adjacent (v_i, v_k) or (v_k, v_j) edges. This method is given in Figure 9. R maintains the relations between each pair of variables, and Q is the queue of updated edges. Since $c+\top = \top$ for all c, we need not compute $R(v_i, v_j)+R(v_j, v_k)$ unless both $R(v_i, v_j)$ and $R(v_j, v_k)$ are not $\top$. *Adj* holds, for each vertex v_i, the set of adjacent vertices v_k such that $R(v_i, v_k) \neq \top$. When an edge (v_i, v_j) is changed, we need only test elements in $Adj(v_i)$ and $Adj(v_j)$. We must, however, ensure *Adj* is updated whenever an edge ceases to be $\top$ – this is done in update_rel. *Adj* can be maintained in constant time with $O(n^2)$ space and initialization time. As mentioned, we do not need to keep track of both an edge and its inverse; we similarly avoid adding (v_j, v_i) to the queue if (v_i, v_j) has already been added.

This procedure is sound, as each step in the algorithm is the application of an inference of the form $x - z \sqsubseteq (x - y) + (y - z)$.

Theorem 2. The procedure wrapdiff_fixpoint terminates.

Proof. Using either the disjoint-set lattice or the interval over-approximation, the $\sqcap$ operation is monotone (according to the $\sqsubseteq$ and $\preceq$ orderings respectively) and non-increasing. At each step of wrapdiff_fixpoint, either some $R(v_i, v_k)$ or $R(v_k, v_j)$ must decrease, or all entries remain constant and the size of Q decreases. As both the disjoint-set and interval domains are finite, there cannot be any infinite descending chains. Hence wrapdiff_fixpoint must terminate. □

The fixpoint time complexity is clearly bounded by $\mathrm{O}(mn^3)$. However, in practice the algorithm runs much faster; we suspect a tighter bound exists which is not dependent on m.

Proposition 1. In cases where a classical difference constraint solver soundly proves unsatisfiability, wrapdiff_fixpoint also proves unsatisfiability.

Proof. Classical difference constraints are unsatisfiable if there is some cycle $C = [c_1, c_2, \ldots, c_k]$ in the graph, such that $S_C = \sum C > 0$. This conclusion is only sound if there is a corresponding cycle $C' = [-c'_k, \ldots, -c'_2, -c'_1]$ which prevents the cycle from wrapping to 0. Let $S_{C'} = \sum C'$, and let $S_C = pm + r$ such that $r \in [1, m-1]$. The cycle C excludes 0 only if $S_{C'} < (p+1)m$. Note that for each edge $c_i \in C$, $c_i - c'_i \leqslant 0$ (otherwise, the cycle $[c_i, c'_i]$ is trivially unsatisfiable).

Consider the behaviour of wrapdiff_fixpoint on the corresponding constraints $\{[c_1, c'_1]_m, [c_2, c'_2]_m, \ldots, [c_k, c'_k]_m\}$. The interval size $c'_i - c_i$ is non-negative. As $S_{C'} - S_C < m$, the size of each partial-sum interval $|\sum_{i=1}^{j} [c_i, c'_i]_m|$ is less than m, so the interval cannot wrap. Adding all the edges then yields the interval $[S_C, S_{C'}]_m$, which does not contain 0. If $0 \notin \sum_{i=1}^{k} [c_i, c'_i]_m$, we also have $\sum_{i=1}^{k-1} [c_i, c'_i]_m \sqcap [c_k, c'_k]_m = \bot$. ($A \sqcap -B \neq \bot$ means there is some x such that $x \in A, -x \in B$. Therefore $0 = x + (-x) \in A + B$.)

Hence, if a cycle exists which allows classical difference logic to soundly conclude unsatisfiability, wrapdiff_fixpoint will do the same. □

4 Experimental Evaluation

In this section, we evaluate the performance of the two **SMT**-based methods, and the incomplete shortest-path approach. For the $\mathbf{SMT}(\mathcal{BV})$ approach, we used the STP solver [6]; for the $\mathbf{SMT}(\mathcal{DL})$ encoding, we used the Z3 theorem prover [3]. The shortest-path algorithm is implemented in C++. The evaluation was conducted on a 3.00GHz Core2 Duo with 2Gb of RAM running Ubuntu GNU/Linux 10.04. Reported times are in milliseconds.

We compared the performance of the two approaches on a set of randomly generated problems over $\mathbb{Z}_{2^{32}}$ with an increasing number of variables. 100 instances were generated for each problem size between 20 and 200 variables. To

Table 1. Comparing the $\mathbf{SMT}(\mathcal{BV})$ and $\mathbf{SMT}(\mathcal{DL})$ approaches with wrapdiff_fixpoint. Time reported (in milliseconds) is the average runtime over 100 instances of each size.

$\|V\|$	$\|C\|$	$\text{TIME}_{\mathcal{BV}}$	$\text{TIME}_{\mathcal{DL}}$	TIME_{fix}	#U	#FP
20	24	50.8	19.2	0.2	24	1
40	48	99.9	24.4	0.4	22	1
60	72	150.0	29.8	0.8	22	1
80	96	197.5	36.4	1.1	29	1
100	120	268.9	43.3	1.7	22	0
120	144	341.3	50.9	2.0	21	0
140	168	404.0	59.0	2.6	22	1
160	192	494.9	65.9	2.8	27	0
180	216	537.7	73.2	3.4	31	1
200	240	675.6	85.5	3.9	25	0

ensure a mix of satisfiable and unsatisfiable instances, the number of constraints $|C|$ was fixed to $1.2|V|$. Of these, $\frac{1}{10}$ are ordering constraints, the remainder being uniformly distributed proximity constraints.[4] Results are given in Table 1. $\text{TIME}_{\mathcal{BV}}$, $\text{TIME}_{\mathcal{DL}}$ and TIME_{fix} denote the time for each method to solve all instances of the given size. #U indicates the number of unsatisfiable instances, and #FP the number of instances which the fixpoint-based method incorrectly reported to be satisfiable.

On these instances, the $\mathbf{SMT}(\mathcal{DL})$ encoding is considerably faster than the $\mathbf{SMT}(\mathcal{BV})$ encoding. The incomplete method is generally around 30 times faster than the $\mathbf{SMT}(\mathcal{DL})$ method, while having a very low false positive rate.

5 Conclusion

Difference logic is useful for program verification and analysis. However, for machine-arithmetic-aware program analysis and verification, classical difference logic is unsound. We have shown that, when extended to modular arithmetic, difference constraints are NP-complete even for $\mathbb{Z}_3$. We have presented two complete methods based on **SMT** techniques, and a sound heuristic based on a fixpoint computation. The heuristic runs substantially faster than the complete methods, and correctly determines unsatisfiability for the majority of the random instances we tested. It would be interesting to develop alternative techniques which improve precision without sacrificing performance. Further work will involve embedding this method in an abstract interpretation framework for static analysis.

Acknowledgments. This work was supported through ARC grant DP110102579. We are grateful to the anonymous reviewers who identified a number of critical misprints in the draft version and suggested an improved SMT encoding which we have adopted.

[4] The solver and instances are available at `ww2.cs.mu.oz.au/~ggange/moddiff/`

References

1. Bjørner, N., Blass, A., Gurevich, Y., Musuvathi, M.: Modular difference logic is hard, arXiv:0811.0987v1 (November 2008) (unpublished)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press (2009)
3. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 179–212. Springer, Heidelberg (1990)
5. Floyd, R.W.: Algorithm 97: Shortest path. Comm. ACM 5, 345 (1962)
6. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
7. Garey, M.R., Johnson, D.S.: Computers and Intractability. Freeman (1979)
8. Gotlieb, A., Leconte, M., Marre, B.: Constraint solving on modular integers. In: Proc. Ninth Int. Workshop Constraint Modelling and Reformulation (2010), `http://www.it.uu.se/research/group/astra/ModRef10/programme.html`
9. John, A.K., Chakraborty, S.: A quantifier elimination algorithm for linear modular equations and disequations. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 486–503. Springer, Heidelberg (2011)
10. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer (2008)
11. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
12. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. J. Automata, Languages and Combinatorics 7(3), 321–350 (2002)
13. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. ACM Trans. Programming Languages and Systems 29(5), Article 29 (2007)
14. Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Signedness-agnostic program analysis: Precise integer bounds for low-level code. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 115–130. Springer, Heidelberg (2012)
15. Regehr, J., Duongsaa, U.: Deriving abstract transfer functions for analyzing embedded software. In: LCTES 2006: Proc. Conf. Language, Compilers, and Tool Support for Embedded Systems, pp. 34–43. ACM Press (2006)
16. Sen, R., Srikant, Y.N.: Executable analysis using abstract interpretation with circular linear progressions. In: Proc. Fifth IEEE/ACM Int. Conf. Formal Methods and Models for Codesign, pp. 39–48. IEEE (2007)
17. Simon, A., King, A.: Taming the wrapping of integer arithmetic. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 121–136. Springer, Heidelberg (2007)
18. Strichman, O., Seshia, S.A., Bryant, R.E.: Deciding separation formulas with SAT. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 209–222. Springer, Heidelberg (2002)
19. Wang, C., Ivančić, F., Ganai, M.K., Gupta, A.: Deciding separation logic formulae by SAT and incremental negative cycle elimination. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 322–336. Springer, Heidelberg (2005)

Asymmetric Unification: A New Unification Paradigm for Cryptographic Protocol Analysis*,**

Serdar Erbatur[1], Santiago Escobar[3], Deepak Kapur[4], Zhiqiang Liu[5], Christopher A. Lynch[5], Catherine Meadows[6], José Meseguer[7], Paliath Narendran[2], Sonia Santiago[3], and Ralf Sasse[8]

[1] Università degli Studi di Verona
serdar.erbatur@gmail.com
[2] University at Albany-SUNY, Albany, NY, USA
dran@cs.albany.edu
[3] DSIC-ELP, Universitat Politècnica de València, Spain
sescobar@dsic.upv.es, ssantiago@dsic.upv.es
[4] University of New Mexico, Albuquerque, NM, USA
kapur@cs.unm.edu
[5] Clarkson University, Potsdam, NY, USA
liuzh@clarkson.edu, clynch@clarkson.edu
[6] Naval Research Laboratory, Washington DC, USA
meadows@itd.nrl.navy.mil
[7] University of Illinois at Urbana-Champaign, USA
meseguer@illinois.edu
[8] Institute of Information Security, ETH Zurich, Switzerland
ralf.sasse@inf.ethz.ch

Abstract. We present a new paradigm for unification arising out of a technique commonly used in cryptographic protocol analysis tools that employ unification modulo equational theories. This paradigm relies on: (i) a decomposition of an equational theory into (R, E) where R is confluent, terminating, and coherent modulo E, and (ii) on reducing unification problems to a set of problems $s =^? t$ under the constraint that t remains R/E-*irreducible*. We call this method *asymmetric unification*. We first present a general-purpose generic asymmetric unification algorithm. and then outline an approach for converting special-purpose conventional

* S. Escobar and S. Santiago were partially supported by EU (FEDER) and the Spanish MEC/MICINN under grant TIN 2010-21062-C02-02, and by Generalitat Valenciana PROMETEO2011/052. The following authors were partially supported by NSF: S. Escobar, J. Meseguer, and R. Sasse under CNS 09-04749 and CCF 09-05584; D. Kapur under CNS 09-05222; C. Lynch, Z. Liu, and C. Meadows under CNS 09-05378, and P. Narendran and S. Erbatur under CNS 09-05286. Part of the S. Erbatur's work was supported while with the Department of Computer Science, University at Albany, and part of R. Sasse's work was supported while with the Department of Computer Science, University of Illinois at Urbana-Champaign. Portions of this paper appeared in an abstract in the informal proceedings of UNIF'11.

** One of the authors is an employee of the US government, and the rights of this work are transferred to the extent transferable according to title 17 U.S.C. 105.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 231–248, 2013.

unification algorithms to asymmetric ones, demonstrating it for *exclusive-or* with uninterpreted function symbols. We demonstrate how asymmetric unification can improve performanceby running the algorithm on a set of benchmark problems. We also give results on the complexity and decidability of asymmetric unification.

1 Introduction

The symbolic analysis of cryptographic protocols has been one of the most successful applications of model-checking to security. In such an analysis, messages are symbolic terms constructed out of function symbols and variables. Message terms often satisfy some equational properties: e.g. that decryption with a key cancels out encryption with the same key or that a symbol satisfies exclusive-or properties. Also, the network is assumed to be under the control of a hostile intruder who can read and modify all traffic, perform any operation available to a legitimate principal, and may be in league with a set of corrupted principals, and thus have access to their keys.

Protocol execution paths are usually computed by unifying messages received with messages sent. Since equational properties are usually involved, the unification must be *modulo* the equational theory describing those properties. The following strategy to achieve unification in protocol analysis, which we call *variant-based unification,* is used in one form or another by many cryptographic protocol analysis tools, including ProVerif [3], OFMC [2], Maude-NPA [7] and Tamarin [17] (see [7] for a detailed comparison). The equational theory is decomposed into (R, E), where R is a set of sort-decreasing rewrite rules that are confluent, terminating, and coherent modulo E (discussed further in Section 2). Given two terms m_1 and m_2 to be unified, complete sets of *irreducible variants* of m_1 and m_2 with respect to (R, E) are computed,[1] and each irreducible variant of m_1 is E-unified with each irreducible variant of m_2. Any unifier that results in either side of the equation being reducible using R modulo E is discarded as redundant. If the complete set of irreducible variants is guaranteed to be finite (that is, (R, E) has the *finite variant property* [5]), this gives a finitary unification procedure [9].

Example 1. Let us consider the following equational theory (Σ, E, R) for the exclusive-or theory, where R consists of the following equations oriented into rules,[2] and E contains the associativity and commutativity (AC) axioms for $\oplus$:

$$X \oplus 0 = X \qquad X \oplus X = 0 \qquad X \oplus X \oplus Y = Y$$

For term $t = M \oplus M$, $(0, id)$ is the only variant. For term $s = X \oplus Y$, the set of its most general variants is

[1] A set V of term-substitution pairs (u, ρ) is a *complete set of variants* of term t with respect to (R, E) iff for any substitution θ there is a $(u, \rho) \in V$ such that the R/E-canonical form $t\theta{\downarrow_{R/E}}$ of $t\theta$ satisfies: $t\theta{\downarrow_{R/E}} =_E u\rho$ (more in Section 2).

[2] Note that the first two equations are not AC-coherent, but adding the third equation (with variable Y) is sufficient to recover that property (see [20,6]).

$$\{\,(X \oplus Y, id),\\ (Z, \{X \mapsto 0, Y \mapsto Z\}), \qquad (Z, \{X \mapsto Z, Y \mapsto 0\}),\\ (Z, \{X \mapsto Z \oplus U, Y \mapsto U\}), (Z, \{X \mapsto U, Y \mapsto Z \oplus U\}),\\ (0, \{X \mapsto U, Y \mapsto U\}), \qquad (Z_1 \oplus Z_2, \{X \mapsto U \oplus Z_1, Y \mapsto U \oplus Z_2\})\}$$

since any possible variant of s is an instance of one of the terms according to the substitution. For term $u = X \oplus n(A, r)$, the set of its most general variants is

$$\{(X \oplus n(A, r), id), (Z, \{X \mapsto n(A, r) \oplus Z\}), (0, \{X \mapsto n(A, r)\})\}.$$

Now, given the unification problem $Y \oplus n(B, r') = X \oplus n(A, r)$ arising in [7] for a simple protocol, the set of irreducible variants for each side is similar to the variants shown above for term u and the pairwise AC-unification of them gives the following substitutions as solutions to the unification problem:

$$\{X \mapsto n(B, r') \oplus Z, Y \mapsto n(A, r) \oplus Z\}\\ \{X \mapsto n(A, r) \oplus Y \oplus n(B, r')\} \qquad \{Y \mapsto n(B, r') \oplus X \oplus n(A, r)\}\\ \{X \mapsto n(A, r), Y \mapsto n(B, r')\} \qquad \{X \mapsto n(A, r) \oplus Z, Y \mapsto n(B, r') \oplus Z\}$$

However, there is only one most general unifier for the exclusive-or theory, $\{X \mapsto n(A, r) \oplus Y \oplus n(B, r')\}$.

The use of variant-based unification is motivated by two key features. First, it is *theory-generic* and can be applied to many of the theories and combinations of theories that arise in cryptographic protocol analysis. Second, it makes possible many *state space reduction techniques* common in cryptographic protocol analysis tools that require messages to be in *irreducible form*. This is the case, for example, when states in which certain subterm patterns appear are discarded. For example, Maude-NPA discards as unreachable any state in which the intruder learns a term containing a nonce before that nonce is generated. Consider a case, discussed in [7] in which the term learned is of the form $n(A, r) \oplus X$, where $\oplus$ satisfies the equational theory of exclusive-or and $n(A, r)$ is a nonce. If X is instantiated to $n(A, r)$ later in the search, the term reduces to 0, but variable X may appear in other positions so that the nonce could not have been generated, making this instantiation impossible; this is represented in our approach as an irreducibility constraint.

Such a strategy, although it has clear advantages, introduces performance costs due to the fact that the attempt to unify each pair of generated irreducible variants can lead to inefficiency, both because of the time it takes to generate all irreducible variants of both terms and because the size of the most general set of unifiers may be larger than optimal, as shown in Example 1. The latter also causes the state space to be larger than expected, since each produced unifier generally results in the creation of a new state. However, it may be possible to relax the irreducibility conditions on messages. For example, Maude-NPA only requires *received* messages to be in irreducible form. This led to the formulation in [7] of the concept of *contextual symbolic reachability analysis* in which irreducible variants, together with associated irreducibility constraints, are computed on only some of the terms appearing in a state. In [7] this was proved sound

and complete with respect to *state reachability analysis achieved via equational unification.*

However, contextual symbolic reachability analysis opens up a new problem: how best to unify two terms, one of which must satisfy an irreducibility constraint.[3] Indeed, the only instance of an asymmetric unification algorithm we could find was a modified variant-based unification, called *asymmetric variant-based unification*, which is similar to variant-based unification described above except that no variant is computed for the side with an irreducibility constraint.

Example 2. Following Example 1, for the *asymmetric* unification problem $Y \oplus n(B, r') = X \oplus n(A, r)$ where $X \oplus n(A, r)$ is irreducible, the solutions computed by asymmetric variant-based unification are:

$$\{X \mapsto n(B, r') \oplus Z, Y \mapsto n(A, r) \oplus Z\} \quad \{Y \mapsto n(B, r') \oplus X \oplus n(A, r)\}$$

However, there is only one most general asymmetric unifier for the exclusive-or theory: $\{Y \mapsto n(B, r') \oplus X \oplus n(A, r)\}$.

This problem, which we call *asymmetric unification* has, to the best of our knowledge, not been investigated before. Thus we ask the question: Is it possible to find asymmetric unification algorithms that can be used in cryptographic protocol analysis and are more efficient than asymmetric variant-based unification?

With this question in mind, we study asymmetric unification as a problem in its own right. After some preliminaries necessary to understanding the paper in Section 2, Section 3 gives a formal definition of asymmetric unification and shows its relation to variant-based unification. Section 4 outlines a general procedure for converting a symmetric algorithm to an asymmetric one, and applies it to exclusive-or with uninterpreted function symbols. In Section 5 we study the complexity and decidability of asymmetric unification, and show there are theories for which symmetric unification is decidable and asymmetric unification is undecidable. Section 6 gives some experimental results on an implementation of this algorithm for asymmetric exclusive-or in Maude-NPA, comparing its performance with the asymmetric variant-based unification, and provides evidence that variant-based unification is far from optimally efficient but theory-generic. Section 7 concludes the paper and discusses future work.

2 Preliminaries

We follow the classical notation and terminology from [19] for term rewriting, and from [16] for rewriting logic and order-sorted notions. We assume an order-sorted signature $\mathbf{\Sigma} = (S, \leq, \Sigma)$ with poset of sorts $(S, \leq)$. We also assume an S-sorted family $\mathcal{X} = \{\mathcal{X}_\mathsf{s}\}_{\mathsf{s} \in \mathsf{S}}$ of disjoint variable sets with each $\mathcal{X}_\mathsf{s}$ countably infinite. $\mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ is the set of terms of sort s, and $\mathcal{T}_{\Sigma,\mathsf{s}}$ is the set of ground terms of sort s. We write $\mathcal{T}_\Sigma(\mathcal{X})$ and $\mathcal{T}_\Sigma$ for the corresponding order-sorted

[3] Note that an irreducibility constraint on a term s that that does appear in the unification problem can be made part of the problem by adding the equation $s = s$.

term algebras. For a term t, $Var(t)$ denotes the set of variables in t. A *substitution* $\sigma \in \mathcal{S}ubst(\Sigma, \mathcal{X})$ is a sorted mapping from a finite subset of $\mathcal{X}$ to $\mathcal{T}_\Sigma(\mathcal{X})$. Substitutions are written as $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where the domain of σ is $Dom(\sigma) = \{X_1, \ldots, X_n\}$ and the set of variables introduced by terms $t_1, \ldots, t_n$ is written $Ran(\sigma)$. The identity substitution is *id*. Substitutions are homomorphically extended to $\mathcal{T}_\Sigma(\mathcal{X})$. The application of a substitution σ to a term t is denoted by $t\sigma$. A *Σ-equation* is an unoriented pair $t = t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ for some sort $\mathsf{s} \in \mathsf{S}$. An *equational theory* (Σ, E) is a pair with Σ an order-sorted signature and E a set of Σ-equations. The *E-subsumption* preorder $t \sqsupseteq_E t'$ (meaning that t is *more general* than t' modulo E) holds between terms $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$ iff there is a substitution σ such that $t\sigma =_E t'$; such a substitution σ is called an *E-match* from t' to t. For substitutions σ, ρ and a set of variables V we define $\sigma =_E \rho$ (over V) if $x\sigma =_E x\rho$ for all $x \in V$; and $\sigma \sqsupseteq_E \rho$ (over V) if there is a substitution η such that $(\sigma\eta)|_V =_E \rho|_V$. We say σ is *equivalent to* ρ if $\sigma \sqsubseteq_E \rho$ and $\rho \sqsubseteq_E \sigma$. An *$E$-unifier* for a Σ-equation $t = t'$ is a substitution σ such that $t\sigma =_E t'\sigma$. For $Var(t) \cup Var(t') \subseteq W$, a set of substitutions $CSU_E^W(t = t')$ is said to be a *complete* set of unifiers for the equality $t = t'$ modulo E away from W iff: (i) each $\sigma \in CSU_E^W(t = t')$ is an E-unifier of $t = t'$; (ii) for any E-unifier ρ of $t = t'$ there is a $\sigma \in CSU_E^W(t = t')$ such that $\sigma|_W \sqsupseteq_E \rho|_W$ (i.e., there is a substitution η such that $(\sigma\eta)|_W =_E \rho|_W$); and (iii) for all $\sigma \in CSU_E^W(t = t')$, $Dom(\sigma) \subseteq (Var(t) \cup Var(t'))$ and $Ran(\sigma) \cap W = \emptyset$.

A *rewrite rule* is an oriented pair $l \to r$, where $l \notin \mathcal{X}$ and $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ for some sort $\mathsf{s} \in \mathsf{S}$. An *(unconditional) order-sorted rewrite theory* is a triple (Σ, E, R) with Σ an order-sorted signature, E a set of Σ-equations, and R a set of rewrite rules. The rewriting relation on $\mathcal{T}_\Sigma(\mathcal{X})$, written $t \to_R t'$ or $t \to_{p,R} t'$ holds between t and t' iff there exist $p \in Pos_\Sigma(t)$, $l \to r \in R$ and a substitution σ, such that $t|_p = l\sigma$, and $t' = t[r\sigma]_p$. The relation $\to_{R/E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $=_E; \to_R; =_E$. A relation $\to_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is defined as: $t \to_{p,R,E} t'$ (or just $t \to_{R,E} t'$) iff there is a non-variable position $p \in Pos_\Sigma(t)$, a rule $l \to r$ in R, and a substitution σ such that $t|_p =_E l\sigma$ and $t' = t[r\sigma]_p$. The transitive (resp. transitive and reflexive) closure of $\to_{R,E}$ is denoted $\to^+_{R,E}$ (resp. $\to^*_{R,E}$). A term t is called $\to_{R,E}$-irreducible (or just R, E-irreducible) if there is no term t' such that $t \to_{R,E} t'$. For $\to_{R,E}$ confluent and terminating, the irreducible version of a term t is denoted by $t{\downarrow_{R,E}}$. In order to guarantee soundness and completeness of $\to_{R/E}$-reducibility by $\to_{R,E}$-reducibility, we require R to be a set of rewrite rules that are: (i) sort-decreasing, (ii) confluent, (iii) terminating, and (iv) coherent modulo E (see [12,20,6]). We call (Σ, E, R) a *decomposition* of an order-sorted equational theory (Σ, G) if $G = R \uplus E$ and R and E satisfy the four conditions above. Given a decomposition (Σ, E, R) of an equational theory, (t', θ) is an *R, E-variant* [9] (or just a variant) of term t iff $t\theta{\downarrow_{R,E}} =_E t'$ and $\theta{\downarrow_{R,E}} =_E \theta$. A decomposition (Σ, E, R) has the *finite variant property* [9] (also called a *finite variant decomposition*) iff for each Σ-term t, a complete set of its most general variants is finite (see Example 1 for a complete set of variants for terms $M \oplus M$ and $X \oplus Y$).

3 Asymmetric Unification

We give a formal definition of asymmetric unification.

Definition 1 (Asymmetric Unification). *Given a decomposition* (Σ, E, R) *of an equational theory* $(\Sigma, E \cup R)$*, a substitution* σ *is an* asymmetric R, E-unifier *of a set* P *of* asymmetric equations $\{t_1 =_{\downarrow} t'_1, \ldots, t_n =_{\downarrow} t'_n\}$ *iff for each asymmetric equation* $t_i =_{\downarrow} t'_i$ *in* P*,* σ *is an* $(E \cup R)$*-unifier of the equation* $t_i = t'_i$ *and* $(t'_i{\downarrow_{R,E}})\sigma$ *is in* R,E*-normal form. A set of substitutions* Ω *is a* complete set of asymmetric R,E-unifiers *of* P *iff: (i) every member of* Ω *is an asymmetric* R,E*-unifier of* P*, and (ii) for every asymmetric* R,E*-unifier* θ *of* P *there exists a* $\sigma \in \Omega$ *such that* $\sigma \sqsupseteq_E \theta$ *(over* $Var(P)$*).*

In the following, we always assume that in every asymmetric equation $t =_{\downarrow} t'$, t' is in normal form; otherwise, we can always normalize t'.

Example 3. Consider the asymmetric unification problem $Y \oplus n(B, r') =_{\downarrow} X \oplus n(A, r)$ arising in [7] for a simple protocol demonstrating the usefulness of the contextual symbolic reachability analysis framework. Then, there is a most general $\oplus$-unifier $X \mapsto Y \oplus n(B, r') \oplus n(A, r)$. However, this is not an asymmetric unifier; but an equivalent $\oplus$-unifier is $Y \mapsto X \oplus n(B, r') \oplus n(A, r)$, which is the singleton most general asymmetric unifier.

For any $(E \cup R)$-unifier θ of P and substitution τ, $\theta\tau$ is also an $(E \cup R)$-unifier of P. But this is not necessarily the case for asymmetric R,E-unifiers.

Example 4. Consider Example 3 and the most general exclusive-or asymmetric unifier $Y \mapsto X \oplus n(B, r') \oplus n(A, r)$. If we apply the substitution $X \mapsto n(A, r)$ to the above unifier, the resulting substitution is no longer an asymmetric unifier of the original asymmetric unification problem.

The question now arises of how to produce such asymmetric algorithms that improve upon the generic variant-based algorithm described above. We discuss one such approach in the next section.

4 An Asymmetric Unification Algorithm for the Theory of Exclusive OR with Uninterpreted Function Symbols

There are two metrics to be considered when optimizing asymmetric unification algorithms for cryptographic protocol analysis. One of course is speed of execution. The other is the size of the most general set of unifiers. Each such unifier results in the production of a new state, so minimizing the size of this set helps to keep the size of the state space down.

One way of minimizing both execution time and mgu size is to convert a symmetric algorithm that has already been optimized for these features. In that case, we need to keep unifiers produced by the original algorithm whenever possible.

We outline a general approach and illustrate it for exclusive-or of Example 1 together with uninterpreted function symbols, chosen because it is the simplest theory appearing in cryptographic protocol analysis that combines both cancellation rules and a non-trivial theory E in the decomposition (Σ, E, R).

Given a decomposition (Σ, E, R) of (Σ, G), and an asymmetric unification problem $\Gamma = \{t_1 =_{\downarrow} t'_1, \ldots, t_n =_{\downarrow} t'_n\}$, the key steps of the approach are:

1. First compute a complete finite set S of G-unifiers using a finitary unification algorithm for G. If S is empty, then there are no asymmetric unifiers.
2. For each such unifier σ from the previous step, check whether every $t'_i\sigma$ is in R,E-normal form. All such unifiers are retained also as asymmetric unifiers.
3. For a unifier σ such that some $t'_i\sigma$ is not in R,E-normal form, compute an equivalent asymmetric unifier if possible.
4. If both of the previous steps fail, this implies that σ or its equivalents cannot be asymmetric unifiers in their full generality. However, there may be some instances obtained by instantiating variables in them which are asymmetric unifiers. A complete set of instances of a given unifier is generated by suitably instantiating variables. This step may be expensive, so it is employed only as a last resort (as demonstrated in Table 4 of Section 6 using unification problems manually chosen to stress this point). For each such instance the above steps are repeated.

We explain below how steps (1)–(4) yield an asymmetric unification algorithm for exclusive or with uninterpreted symbols (XOR) from a symmetric one. Variables appearing in Γ are called *original variables* to distinguish them from new variables, called *support variables* by the inference rules. For a unification problem Γ and an XOR unifier σ we say in the assignment $x \mapsto t \oplus T \in \sigma$ some original variable x has a *conflict* at some simple term t if

- there exists $u =_{\downarrow} v[x \oplus s] \in \Gamma$ and
- there exists T' such that $s\sigma = t \oplus T'$

where s and t are *simple terms* (i.e., a term that does not have $\oplus$ as its outermost symbol) and T' might be empty. The significance of conflicts is that a substitution of x cannot include t as a subterm, in order to ensure the irreducibility of the right side of equations in Γ.

We present the algorithm as a collection of inference rules on a triple of sets:

$$\frac{\sigma \| \Upsilon \| \Delta}{\sigma' \| \Upsilon' \| \Delta'},$$

where σ is an XOR unifier of Γ, Υ is a set of *constraint pairs* in which each member has the form (v, s) (to mean that a substitution of v cannot include s as a subterm to ensure the irreducibility of the right side of equations in Γ) , and Δ is a set of disequations of the form $s \oplus t \neq^? 0$, with s and t having the same topmost uninterpreted function symbol.

A complete set of XOR-unifiers is first generated using an XOR-unification algorithm. For each XOR unifier σ, the algorithm starts with a triple $\sigma \| \emptyset \| \emptyset$.

The algorithm may generate numerous branches, some of which lead to a dead end because either (i) no inference rule is applicable or (ii) the candidate for an *XOR* unifier violates a constraint in the second component or a disequation in the third component. Different branches can generate equivalent asymmetric unifiers or asymmetric unifiers which are instances of other asymmetric unifiers.

We use the following notation. The result of applying a substitution θ to $\Upsilon = \{(v_1, s_1), \cdots, (v_n, s_n)\}$ is $\Upsilon\theta = \{(v_i, s_i\theta \downarrow) | (v_i, s_i) \in \Upsilon\}$; we will rewrite $(v_i, t_1 \oplus \cdots \oplus t_n)$ to $(v_i, t_1), \cdots, (v_i, t_n)$. A substitution δ satisfies Υ iff δ satisfies every constraint pair in Υ, i.e., given a pair $(v, s) \in \Upsilon$, δ satisfies (v, s) iff $\delta(v) \oplus \delta(s)$ is irreducible using R, E (in this case the rules are the theory of XOR from Example 1). If δ does not satisfy Υ, then δ violates Υ. Similarly, δ satisfies Δ iff δ satisfies every disequation $s \oplus t \neq 0 \in \Delta$, in other words $(s\delta \oplus t\delta)$ does not rewrite to 0.

The Inference System

All inference rules below are don't care nondeterministic rules. They are grouped as: **Splitting**, **Branching** and **Instantiation**. The algorithm runs in two phases. In the first phase, the **Splitting** and **Branching** rules are applied, attempting to generate an asymmetric *XOR* unifier equivalent to the original *XOR* unifier. The **Splitting** rule is applied as much as possible to (i) move all toplevel original variables out of the range of an *XOR* unifier, while (ii) eliminating conflicts between original variables and subterms with which they appear in t_i's in Γ. Once it is no longer applicable, an *XOR* unifier equivalent to the original unifier is constructed such that its range only includes new variables at top levels. Then, branching rules are repeatedly applied attempting to eliminate conflicts between support variables with other variables and nonvariable subterms The **Non-Variable Branching** rule, which eliminates a conflict between a support variable and a nonvariable subterm, is repeatedly applied first. This is followed by (i) the **Auxiliary Branching** rule and (ii) the **Variable Branching** rule. The last two rules may not eliminate any conflicts; however they are helpful later during the second phase. In this first phase, if any of the branches yields an asymmetric *XOR* unifier, the algorithm terminates; it is not necessary to consider other branches as all asymmetric *XOR* unifiers from various branches are equivalent.

If the first phase does not succeed in generating an equivalent asymmetric *XOR* unifier, all branches generated from the first phase must be considered in the second phase. Instantiation rules are now applied to generate instances of equivalent *XOR* unifiers. The **Decomposition Instantiation** rule generates instances of an *XOR* unifier so that the rules $x \oplus x \oplus y \mapsto y$ and $x \oplus x \mapsto 0$ are applicable, whereas the **Elimination Instantiation** rule generates instances by setting support some variables to 0. It is possible that an *XOR* unifier generated by the **Elimination Instantiation** rule is equivalent to the original *XOR* unifier (since it may have been generated by instantiating a support variable to 0 implying that it was unnecessary to introduce that support variable).

If along a branch, a result of **Decomposition Instantiation** is not an asymmetric *XOR* unifier, the algorithm moves again to the first phase and applies **Splitting**, since some of the original variables underneath interpreted function symbols may get elevated to the top level in substitutions of original variables. **Elimination Instantiation** is repeatedly applied only after **Decomposition** cannot be applied any further. If the result is not an asymmetric *XOR* unifier, then the **Branching** rules are applied by returning to the first phase (**Splitting** is not applicable in this case).

The Splitting Rule

This rule transforms an *XOR* unifier σ into an equivalent *XOR* unifier σ' such that all the top variables in $Range(\sigma')$ are support variables.

$$\frac{[x \mapsto y \oplus S \oplus T] \cup \sigma \| \Upsilon \| \Delta}{([x \mapsto y \oplus S \oplus T] \cup \sigma) \circ \theta \| \Upsilon\theta \| \Delta\theta}$$

where $\theta = \{y \mapsto v \oplus S\}$ and v is a fresh support variable. The rule is applied only if (i) $x, y \in Vars(\Gamma)$ and (ii) $y \notin Vars(S)$.

Even though S and T can be chosen in any way, if x has a conflict at some simple term s in $S \oplus T$, then for efficiency in our implementation, we will put s into S, unless $y \in Vars(s)$. After **Splitting** there will be no top level original variables in the range of σ. So from now on, we assume that all the top variables which appear in the range of σ are support variables.

The Branching Rules

The main objective in applying the two branching rules is to try to transform an *XOR* unifier into an equivalent one without conflicts.

Non-Variable Branching. This rule considers the case that some original variable x has a conflict at some non-variable simple term s.

$$\frac{\sigma \| \Upsilon \| \Delta}{\sigma \circ \theta \| (\Upsilon[v'/v] \cup (v', s))\theta \| \Delta\theta \quad \bigvee \quad \sigma \| \Upsilon \cup \{(v, s)\} \| \Delta\theta}$$

where there exists an assignment $[x \mapsto v \oplus s \oplus S] \in \sigma$ and $\theta = [v \mapsto v' \oplus s]$ with v' being a fresh support variable, under the conditions that x has a conflict at a simple nonvariable terms s in Γ where (i) $v \notin Vars(s)$ and (ii) $(v, s) \notin \Upsilon$.

Above, $\Upsilon[v'/v]$ means: replace all occurrences of the variable v in the first component of every pair in Υ by the variable v'. The first branch is used when the conflict between x and s is successfully resolved using v by introducing a new support variable v'; the second branch is used when that is not possible, thus leading to an additional constraint (v, s) implying that v and s are in conflict.

Auxiliary Branching. This rule is applied when an original variable conflict with another original variable in Γ and their substitutions in an XOR unifier share a common part.

$$\frac{\sigma \| \Upsilon \| \Delta}{\sigma \circ \theta \| (\Upsilon[v'/v] \cup (v', s))\theta \| \Delta\theta \quad \bigvee \quad \sigma \| \Upsilon \cup \{(v, s)\} \| \Delta}$$

where $\theta = \{v \mapsto v' \oplus s\}$ with v' being a fresh support variable, and there exist two assignments $[x \mapsto v \oplus s \oplus S, y \mapsto v \oplus S']$ in σ. This rule is applied only if (i) x, y are in conflict in Γ, (ii) s is a simple non-variable term and $v \notin Vars(s)$ and (iii) $(v, s) \notin \Upsilon$.

The additional simple nonvariable term s in the substitution for x in an XOR unifier is used to possibly eliminate the conflict with a new variable v', which stands for the common shared part of x and y. The reader will notice that unlike the **Non-Variable Branching** rule, both branches after this rule still have conflicts in the substitutions of x and y which are in conflict in Γ. So this rule does not solve the conflict directly; it is preparing for the instantiation part.

Variable Branching. This rule is similar to the **Auxiliary Branching** rule and is applied when two original variables x and y have a conflict in Γ and share a common support variable v_1 in their substitutions in an XOR unifier. The key difference from the **Auxiliary Branching** rule is that instead of the substitution for x having a simple nonvariable term that is not in conflict with v_1, it has another support variable v_2. The common support variable v_1 is then split into two parts: the common part of x and y, represented by v_{12}, and the remaining parts of x and y, represented by v_1' and v_2', respectively.

$$\frac{\sigma \| \Upsilon \| \Delta}{\sigma \circ \theta \| \Upsilon'\theta \| \Delta\theta \quad \bigvee \quad \sigma \| \Upsilon \cup \{(v_1, v_2)\} \| \Delta}$$

where σ includes $[x \mapsto v_1 \oplus v_2 \oplus S, y \mapsto v_1 \oplus S']$, $\theta = [v_1 \mapsto v_{12} \oplus v_1', v_2 \mapsto v_{12} \oplus v_2']$, v_{12}, v_1' and v_2' are fresh support variables, and $\Upsilon' = (\Upsilon[v_{12}/v_1][v_{12}/v_2] \cup \Upsilon[v_1'/v_1] \cup \Upsilon[v_2'/v_2] \cup \{(v_{12}, v_1'), (v_{12}, v_2'), (v_1', v_2'), (v_1', v_{12}), (v_2', v_{12}), (v_2', v_1')\}$. This rule is applied only if (i) x and y have a conflict in Γ and (ii) $(v_1, v_2) \notin \Upsilon$.

The first branch is the case when v_1 and v_2 have a common part, whereas the second branch is the case when v_1 and v_2 have nothing in common.

Instantiation Rules

The following instantiation rules are used for solving conflicts by instantiating support variables based on the equations $x + x \to 0$ and $x + 0 \to x$

Decomposition Instantiation. This rule is used to solve the case that some original variable x has a conflict with a simple nonvariable term t.

$$\frac{\sigma \| \Upsilon \| \Delta}{\sigma \circ \theta_1 \| \Upsilon\theta_1 \| \Delta\theta_1 \quad \bigvee \cdots \bigvee \quad \| \sigma \circ \theta_n \| \Upsilon\theta_n \| \Delta\theta_n \quad \bigvee \quad \sigma \| \Upsilon \| \Delta''}$$

where there exists an assignment $[x \mapsto s \oplus t \oplus S]$ in σ, x has a conflict with a simple nonvariable subterm s in Γ and s and t have the same topmost uninterpreted symbol; $\{\theta_1, \cdots, \theta_n\}$ is a complete set of *XOR* unifiers of $s \stackrel{?}{=} t$ and $\Delta'' = \Delta \cup \{s \oplus t \neq^? 0\}$.

Elimination Instantiation. This rule is used to solve the case that some original variable x has a conflict at some support variable v.

$$\frac{[x \mapsto v \oplus S] \cup \sigma \| \Upsilon \| \Delta}{([x \mapsto S] \cup \sigma) \circ \theta \| \Upsilon\theta \| \Delta\theta}$$

where $\theta = \{v \mapsto 0\}$, x and y are in conflict in Γ for some y. The rule is applied only if $y\sigma = v \oplus S'$ with S' having at least one subterm.

Because v maps to 0, all pairs (v, s) in Υ will be removed from Υ.

Theorem 1. *The asymmetric unification algorithm described above is sound, terminating, and complete.*

Proof. Soundness easy to establish since we need to show that if an inference rule generates an asymmetric *XOR* unifier, then that unifier is either equivalent to an *XOR* unifier or an instance of an *XOR* unifier. Termination and completeness are nontrivial. We sketch the proofs below; detailed proofs are given in [14].

For termination, we must prove that the algorithm does not go into cycles or keep on introducing new variables in the first phase; the termination of the second phase is easy to establish. The intertwining of two phases also terminates if it can be proved that throughout the algorithm, only a bounded number of new variables are introduced by various rules. Only the Splitting and Branching rules introduce new variables. We thus first prove that they are applied only finitely often. We then complete the proof of the absence of cycles by proving that the Instantiation rules are applied only finitely often.

Intuitively, the number of new variables generated is bounded by (i) the number of all possible subsets of nonvariable subterms in the original problem and (ii) an original variable sharing exclusively with another original variable, two original variables, and so on. The substitution for any original variable x is an *XOR* of (i) a subset of nonvariable subterms appearing in the original problem and their instances due to the Decomposition Instantiation Rule, (ii) original variables with which x has no conflict and (ii) new variables standing for disjoint subsets of original subterms in the substitution of x different from substitutions of variables in conflict with x (much like v_{12}, the common part of x and y, and v'_1 and v'_2, the parts of x and y that are disjoint from each other in the Variable Branching rule). New variables also serve as placeholders to allow for generation of conflict-free instances of an *XOR* unifier in case that it does not have an equivalent asymmetric *XOR* unifier.

Once it is proved that the algorithm only introduces finitely many new variables (thus implying that the Splitting rule and the three Branching rules are only applied finitely many times), the proof of termination becomes easier since it only needs to be made sure that the two instantiation rules cannot be applied

infinitely often. The Elimination Instantiation rule reduces the size of the triple since variables get instantiated to 0 and then simplified.

The Decomposition Instantiation rule reduces the number of simple terms in the substitutions for the original variables along the branch due to the unification of s, t in $x \mapsto s \oplus t \oplus S$ thus replacing $s \oplus t \oplus S$ by $\theta_i(S)$. For the branch in which the disequation $s \oplus t \neq^? 0$ is added, the set of instances of the original *XOR* unifier being investigated get reduced[4].

To prove completeness we must show that every inference rule only prunes those non-asymmetric instances of an *XOR* unifier. Discarding of instances of an *XOR* unifier can take place only with the instantiation rules. The Decomposition Instantiation rule does not discard any instances of an *XOR* unifier since the branching is done based on whether two nonvariable subterms s and t are *XOR* unifiable or not. The Elimination Instantiation rule discards instances of an *XOR* unifier by considering only the case when a new variable is made equal to 0, while not considering the case when that new variable is not equal to 0, but this is done only if no other way is possible. □

5 Decidability of Asymmetric Unification

It is easy to see that asymmetric R,E-unification is at least as hard as $E \cup R$-unification, since every asymmetric R,E-unifier is also an $E \cup R$-unifier.However, nothing can be said about its asymmetric unifiers of a problem from its set of unifiers. The unification problem could have a nonempty set of unifiers, whereas the asymmetric unification problem need not have any asymmetric unifier. Or, the unification problem could have a single most general unifier, whereas the asymmetric unification problem has exponentially many solutions, as illustrated using the following asymmetric unification problem:

$$x_1 \oplus \ldots \oplus x_n =_{\downarrow} a_1 \oplus \ldots a_n, \quad x_1 \oplus \ldots \oplus x_n =_{\downarrow} x_1 \oplus \ldots \oplus x_n$$

which has a single unifier $x_1 \mapsto x_2 \oplus ... \oplus x_n \oplus a_1 \oplus ... \oplus a_n$, and $n!$ asymmetric unifiers.

We show that there exist theories for which unification is decidable and asymmetric unification is undecidable. These results are obtained by using a restricted version of the Modified Post Correspondence Problem (MPCP) [11, Section 9.4.2]. First, we define the theory $(\Sigma, \mathcal{R}_\mu)$ based on the MPCP version here and prove that unification modulo $\mathcal{R}_\mu$ (and hence asymmetric unification modulo $\mathcal{R}_\mu$) is undecidable by a reduction from MPCP. Moreover, matching modulo $\mathcal{R}_\mu$ is shown to be decidable and finitary. We use these facts to extend $(\Sigma, \mathcal{R}_\mu)$ to a theory for which unification is decidable but asymmetric unification is not.

[4] The set of all possible instances of an *XOR* unifiers which must considered for investigating equivalent asymmetric *XOR* unifiers is finite since original variables only need to be instantiated by an *XOR* of a subset of finitely many nonvariable subterms, variable subterms and new variables.

Let $\Omega = \{a, b\}$, and let $P = \{(\alpha_i, \beta_i) \mid i = 1, \ldots, n\} \subseteq \Omega^+ \times \Omega^+$ be a finite set of pairs of non-empty strings over Σ. Then consider the following restricted version of the Modified Post Correspondence Problem (MPCP) which is undecidable [10, Theorem 4.4]:

Instance: A non-empty string $\alpha \in \Omega^+$.
Question: Does there exist a sequence of indices $i_1, \ldots, i_k \in \{1, \ldots, n\}$ such that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k}\alpha = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$?

We construct $\mathcal{R}_\mu$ from this problem as follows. We start by defining the signature of $\mathcal{R}_\mu$ as $\Omega' = \Omega'_1 \cup \Omega'_3$ where $\Omega'_1 = \{a, b, 1, \ldots, n\}$ and $\Omega'_3 = \{f\}$. Thus Ω' has $n+2$ unary function symbols and one ternary function symbol. Additionally, we convert strings in the MPCP instance to terms as usual. For any string $w \in \Omega^*$, let $\widetilde{w}(x)$ denote the term formed by treating a and b as unary function symbols and the concatenation operator as function composition; in other words,

$$\widetilde{\lambda}(x) = x, \qquad \widetilde{au}(x) = a(\widetilde{u}(x)), \qquad \widetilde{bu}(x) = b(\widetilde{u}(x)).$$

For each pair (α_i, β_i) of the MPCP we create a rule

$$f(x,\, i(y),\, z) \to f(\widetilde{\alpha_i}(x),\, y,\, \widetilde{\beta_i}(z))$$

Let $\mathcal{R}_\mu$ be the set of all such rules, and let Σ be the set of symbols involved in creating them. This system is confluent and terminating: we observe that $\mathcal{R}_\mu$ is left-linear and has no critical pairs, hence is orthogonal. Thus the confluence of the system follows. In addition it is easy to show that $\mathcal{R}_\mu$ is terminating, since each application of rules of $\mathcal{R}_\mu$ decreases the number of occurrences of a symbol $j \in \{1, \ldots, n\}$ in a term. Finally, $(\Sigma, \emptyset, \mathcal{R}_\mu)$ is trivially sort-decreasing and coherent, since all symbols have the same sort, and E is empty. In particular, by the following lemma, every congruence class modulo $\mathcal{R}$ is finite.

Lemma 2. *Let $\mathcal{R}$ be a convergent term rewriting system. If $\mathcal{R}^{-1}$ is terminating then every congruence class modulo $\mathcal{R}$ is finite.*

Lemma 3. *Matching modulo $\mathcal{R}_\mu$ is decidable and finitary.*

Proof. Note that $\mathcal{R}_\mu^{-1}$ is terminating; hence by Lemma 2 for each term s, the congruence class $[s]_{\mathcal{R}_\mu}$ is finite. It was shown by Bürkert, Herold and Schmidt-Schauß [4] that if $\mathcal{R}$ is a theory where every congruence class is finite then the matching problem modulo $\mathcal{R}$ is decidable and is of matching type finitary. □

Lemma 4. *Let c be an arbitrary constant. The following unification problem has a solution if and only if the instance of the MPCP problem has a solution.*

$$f(\alpha(c),\, V,\, c) \;=^?_{\mathcal{R}_\mu}\; f(X,\, c,\, X)$$

Proof. The "if" part is straightforward: assume that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k}\alpha = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$ for some indices $i_1, \ldots, i_k \in \{1, \ldots, n\}$. Then

$$\tau \;=\; \{X \mapsto \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}(c),\; V \mapsto i_k i_{k-1}\ldots i_1(c)\}$$

is a unifier for the unification problem. Note that we have

$$\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k}\alpha(c) \;=\; \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}(c) \quad \text{and thus}$$

$$\begin{aligned} f(\alpha(c),\tau(V),c) &\longrightarrow^{*}_{\mathcal{R}_\mu} f(\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k}\alpha(c), c, \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}(c)) \\ &\equiv \quad f(\alpha(\tau(X)), c, \tau(X)) \end{aligned}$$

Conversely, suppose θ is a solution for the above equation. Then the following necessarily holds: $\theta(f(\alpha(c), V, c)) = f(\alpha(c), \theta(V), c) \longrightarrow^{!}_{\mathcal{R}_\mu} f(\theta(X), c, \theta(X))$. Now a solution for the MPCP instance can be obtained from $\theta(V)$ as follows. Each rewrite step reveals an $i_j \in \{1,\ldots,n\}$ by deleting the top symbol from $\theta(V)$. Otherwise $\mathcal{R}_\mu$ does not apply to $f(\alpha(c), \theta(V), c)$ and hence we conclude that there exists no sequence of $i_1,\ldots,i_k \in \{1,\ldots,n\}$. Thus by using i_j's we form a solution to the MPCP problem. □

We now extend $\mathcal{R}_\mu$ by adding a special constant $\perp$ (annihilator) such that, if it occurs in a term t, then t reduces to $\perp$. That is, we add the rules

$$a(\perp) \to \perp, \qquad b(\perp) \to \perp, \qquad f(x, y, \perp) \to \perp, \qquad f(x, \perp, y) \to \perp,$$

$$f(\perp, x, y) \to \perp, \quad \text{and} \quad i(\perp) \to \perp, \quad i \in \{1, \ldots, n\}$$

Let $\mathcal{R}_\perp$ be the set of those new rules. Then we denote $\mathcal{R} = \mathcal{R}_\mu \cup \mathcal{R}_\perp$ the system extended by annihilator rules. Note that $\mathcal{R}$ is convergent as well.

Since equations where both sides contain variables can be trivially solved by setting the variables to $\perp$, we can show that

Theorem 5. *Unification modulo $\mathcal{R}$ is decidable.*

Proof. Without loss of generality, we may consider a problem consisting of one equation $s =^{?}_{\mathcal{R}} t$. In the case that s or t are ground, the problem reduces to one of matching modulo $\mathcal{R}_\mu$, which is decidable by Lemma 3. In the case that both s and t contain variables, the problem becomes $\perp =^{?}_{\mathcal{R}} \perp$ after substituting $\perp$ to the variables on both sides and reducing. Thus it has a trivial solution. □

Theorem 6. *Asymmetric unification modulo $\mathcal{R}$ is undecidable.*

Proof. Consider the problem $f(\alpha(c), V, c) =^{?}_{\downarrow\mathcal{R}} f(X, c, X)$. A unifier obtained by substituting $\perp$ to the variables on both sides would violate asymmetry. Moreover, it is impossible to obtain a unifier by subsituting $\perp$ to the variables in the left side alone. Thus the problem has an asymmetric unifier modulo $\mathcal{R}$ if and only if it has an asymmetric unifier modulo $\mathcal{R}_\mu$. Since $f(X, c, X)$ is irreducible modulo $\mathcal{R}_\mu$ no matter what substitution is made to X, the problem has an asymmetric unifier modulo $\mathcal{R}_\mu$ if and only if it has a symmetric unifier. The result follows from Lemma 4 and the undecidability of MPCP. □

Table 1. Unification Problems in ESORICS12 protocol

Unif. Problem	T. A-V	# A-V	T. D-A	# D-A	% T.	% #
$NS_1 \oplus NS_2 =\downarrow NS_3 \oplus N_A$	153	12	153	1	0	91
$NS_1 \oplus N_A =\downarrow NS_2 \oplus NS_3$	137	5	121	1	11	80
$NS_1 \oplus NS_2 =\downarrow NS_3 \oplus NS_4 \oplus NS_5$	286	54	116	1	59	98
$NS_1 \oplus NS_2 =\downarrow NS_3 \oplus NS_4 \oplus N_A$	159	36	115	1	27	97
$NS_1 \oplus NS_2 =\downarrow N_A$	127	4	114	1	10	75
$NS_1 \oplus NS_2 =\downarrow null$	128	1	105	1	17	0
$NS_1 \oplus NS_2 =\downarrow null \oplus NS_3$	130	7	105	1	20	85

Table 2. Unification Problems in WEPP protocol

Unif. Problem	T. A-V	# A-V	T. D-A	# D-A	% T.	% #
$M_1 \oplus M_2 =\downarrow M_3 \oplus pair(V_1, M_4)$	51	12	44	1	13	91
$pair(V, rc4(V_1, kAB) \oplus ([N_A, c(N_A)])) =\downarrow pair(V_1, M_1)$	30	1	29	1	3	0
$M_1 \oplus M_2 =\downarrow M_3 \oplus V_1$	33	12	32	1	3	91
$M_1 \oplus M_2 =\downarrow M_3 \oplus ([N_1, c(N_2)])$	34	12	30	1	11	91
$M_1 \oplus M_2 =\downarrow M_3 \oplus pair(V_1, pair(V_2, M_4))$	36	12	30	1	16	91

6 Experiments with Unification Problems Arising in Protocol Analysis

We implemented a variant-based algorithm for XOR and an algorithm produced by applying the procedure outlined in Section 4 to the special-purpose XOR algorithm of [13] in Maude-NPA and experimentally compared their performance. We have run the experiments presented in this Section in an Intel Xeon machine with 4 cores and 24GB of memory, using Maude 2.7, which includes a built-in implementation of the variant generation.

Tables 1, 2 and 3 gather the results of unification problems from the following protocols: (i) the running protocol example of [7], referred as *ESORICS12*, (ii) the Wired Equivalent Privacy Protocol (WEPP) of [1], and (iii) the TMN protocol of [18,15], respectively. Table 4 gathers the results of some more complex problems manually defined by the authors to stress the algorithms. Here each unification problem combines several subproblems, shown below the table. The ESORICS12, WEPP and TMN protocols were used in the experiments performed in [7], in order to compare the contextual symbolic reachability approach presented in that paper with other approaches. However, the experiments presented in this Section are more focused on concrete unification problems that

Table 3. Unification Problems in TMN protocol

Unif. Problem	T. A-V	# A-V	T. D-A	# D-A	% T.	% #
$M_1 \oplus M_2 =\downarrow M_3 \oplus M_4$	115	18	105	1	8	94
$M_1 \oplus M_2 =\downarrow M_3 \oplus M_4 \oplus M_5$	5749	1	74	1	98	0
$M_1 \oplus M_2 =\downarrow M_3 \oplus pair(M_4, M_5)$	71	12	71	1	0	91
$pair(M_1, M_2) =\downarrow pair(M_3, M_4)$	65	1	70	1	-1	0
$M_1 \oplus M_2 =\downarrow pair(M_3, M_4)$	67	4	71	1	0	91
$M_1 \oplus M_2 =\downarrow null \oplus M_3$	66	7	70	1	-6	85

Table 4. Other Unification Problems

Unif. Problem	T. A-V	# A-V	T. D-A	# D-A	% T.	% #
$SP4 \wedge SP1 \wedge SP2$	422	4	68	3	83	25
$SP5 \wedge SP1 \wedge SP2$	408	24	131	7	67	70
$SP6 \wedge SP1 \wedge SP2$	416	100	491	15	-18	85
$SP7 \wedge SP1 \wedge SP2$	454	360	3732	31	-722	91
$SP8 \wedge SP1 \wedge SP2 \wedge SP3$	151387	3	47	1	99	66
$SP9 \wedge SP1 \wedge SP2 \wedge SP3$	153913	33	80	3	99	66
$SP10 \wedge SP1 \wedge SP2 \wedge SP3$	154137	201	157	7	99	96
$SP11 \wedge SP1 \wedge SP2 \wedge SP3$	154534	1053	349	15	99	98
$SP12 \wedge SP1 \wedge SP2 \wedge SP3$	160114	5073	829	31	99	99

SP1 $= M_1 \oplus M_2 =\downarrow M_1 \oplus M_2$
SP2 $= M_1 \oplus M_3 =\downarrow M_1 \oplus M_3$
SP3 $= M_1 \oplus M_4 =\downarrow M_1 \oplus M_4$
SP4 $= M_1 \oplus M_2 \oplus M_3 =\downarrow a \oplus b$
SP5 $= M_1 \oplus M_2 \oplus M_3 =\downarrow a \oplus b \oplus c$
SP6 $= M_1 \oplus M_2 \oplus M_3 =\downarrow a \oplus b \oplus c \oplus d$
SP7 $= M_1 \oplus M_2 \oplus M_3 =\downarrow a \oplus b \oplus c \oplus d \oplus e$
SP8 $= M_1 \oplus M_2 \oplus M_3 \oplus M_4 =\downarrow a$
SP9 $= M_1 \oplus M_2 \oplus M_3 \oplus M_4 =\downarrow a \oplus b$
SP10 $= M_1 \oplus M_2 \oplus M_3 \oplus M_4 =\downarrow a \oplus b \oplus c$
SP11 $= M_1 \oplus M_2 \oplus M_3 \oplus M_4 =\downarrow a \oplus b \oplus c \oplus d$
SP12 $= M_1 \oplus M_2 \oplus M_3 \oplus M_4 =\downarrow a \oplus b \oplus c \oplus d \oplus e$

occur during the analysis of these protocols and the efficiency of asymmetric unification algorithms when solving them in terms of number of unifiers and execution time.

In each table the first and second columns show, respectively, the execution time (in milliseconds) and the number of unifiers obtained using the asymmetric variant-based unification algorithm. The third and fourth columns show, respectively, the execution time (in milliseconds) and the number of unifiers obtained using the special-purpose asymmetric unification algorithm for exclusive-or. Finally, the two last columns present a percentage that reflects the performance improvement of the special-purpose asymmetric unification algorithm with respect to the asymmetric variant-based algorithm in terms of execution time and number of unifiers obtained, respectively.

On the average the special-purpose asymmetric unification algorithm is about 8% faster than the variant-based one, and generates about 71% fewer unifiers. Note, however, that in many cases the reduction in the number of unifiers is more than 90%. Moreover the asymmetric variant-based unification algorithm does not provide a minimal set of unifiers, whereas the special-purpose asymmetric algorithm does in all our examples. Indeed, all the asymmetric unification problems extracted from protocols have a singleton most general asymmetric unifier, as shown in Tables 1, 2, and 3. However, as shown in Table 4, the special-purpose algorithm can sometimes be slower than the variant-based one, even when it generates a smaller most general set of asymmetric unifiers. The reason is that the post-processing step of the algorithm explained in Section 4 in which appropriate asymmetric unifiers are only instances of the computed unifiers is sometimes very expensive.

7 Conclusions and Future Work

We have shown how asymmetric unification arises in a natural way when analyzing cryptographic protocols. We have investigated the complexity and decidability of the problem and shown that variant-based unification can be adapted to

obtain a *theory-generic* asymmetric unification algorithm. We have also outlined an approach for converting symmetric algorithms to asymmetric ones and applied it to an exclusive-or algorithm. Our experimental results are encouraging, not only for increasing speed but for reducing the number of unifiers.

We plan to refine our procedures for converting algorithms by applying them to other theories of interest to cryptographic protocol analysis. We conjecture that our method for converting symmetric algorithms to asymmetric ones can be developed into an algorithm for certain classes of unification algorithms and will investigate this further. We will also investigate *combining* asymmetric algorithms, since combined theories are a common occurrence in cryptographic protocols. Variant-based narrowing lends itself relatively easily to such combination. Special-purpose asymmetric unification algorithms will not be as easy to combine, but we have been investigating combination techniques that take advantage of special properties of the theories of interest to cryptographic protocol analysis and plan to apply them in the asymmetric setting.

References

1. IEEE 802.11 Local and Metropolitan Area Networks: Wireless LAN Medium Access Control (MAC) and Physical (PHY) Specifications (1999)
2. Basin, D., Mödersheim, S., Viganò, L.: An on-the-fly model-checker for security protocol analysis. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 253–270. Springer, Heidelberg (2003)
3. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: CSFW, pp. 82–96. IEEE Computer Society (2001)
4. Bürckert, H.-J., Herold, A., Schmidt-Schauß, M.: On equational theories, unification, and (un)decidability. Journal of Symbolic Computation 8(1/2), 3–49 (1989)
5. Comon-Lundh, H., Delaune, S.: The finite variant property: How to get rid of some algebraic properties. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 294–307. Springer, Heidelberg (2005)
6. Durán, F., Meseguer, J.: A Maude coherence checker tool for conditional order-sorted rewrite theories. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 86–103. Springer, Heidelberg (2010)
7. Erbatur, S., Escobar, S., Kapur, D., Liu, Z., Lynch, C., Meadows, C., Meseguer, J., Narendran, P., Santiago, S., Sasse, R.: Effective symbolic protocol analysis via equational irreducibility conditions. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 73–90. Springer, Heidelberg (2012)
8. Erbatur, S., Escobar, S., Kapur, D., Liu, Z., Lynch, C., Meadows, C., Meseguer, J., Narendran, P., Sasse, R.: Asymmetric unification: A new unification paradigm for cryptographic protocol analysis. In: UNIF 2011 (2011), `https://sites.google.com/a/cs.uni.wroc.pl/unif-2011/program`
9. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. J. Log. Algebr. Program. 81(7-8), 898–928 (2012)
10. Harju, T., Karhumäki, J., Krob, D.: Remarks on generalized post correspondence problem. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046, pp. 39–48. Springer, Heidelberg (1996)
11. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation - international edition, 2nd edn. Addison-Wesley (2003)

12. Jouannaud, J.-P., Kirchner, H.: Completion of a set of rules modulo a set of equations. SIAM J. Comput. 15(4), 1155–1194 (1986)
13. Liu, Z., Lynch, C.: Efficient general unification for XOR with homomorphism. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 407–421. Springer, Heidelberg (2011)
14. Liu, Z.: Dealing Efficiently with Exclusive OR, Abelian Groups and Homomorphism in Cryptographic Protocol Analysis. PhD thesis, Clarkson University (2012), http://people.clarkson.edu/~clynch/papers/Dissertation_of_Zhiqiang_Liu.pdf
15. Lowe, G., Roscoe, A.W.R.: Using CSP to detect errors in the TMN protocol. IEEE Transactions on Software Engineering 23, 659–669 (1997)
16. Meseguer, J.: Conditional rewriting logic as a united model of concurrency. Theor. Comput. Sci. 96(1), 73–155 (1992)
17. Schmidt, B., Meier, S., Cremers, C.J.F., Basin, D.A.: Automated analysis of Diffie-Hellman protocols and advanced security properties. In: Proc. CSF 2012, pp. 78–94. IEEE (2012)
18. Tatebayashi, M., Matsuzaki, N., Newman Jr., D.B.: Key distribution protocol for digital mobile communication systems. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 324–334. Springer, Heidelberg (1990)
19. TeReSe, editor. Term Rewriting Systems. Cambridge University Press (2003)
20. Viry, P.: Equational rules for rewriting logic. Theor. Comp. Sci. 285(2), 487–517 (2002)

Hierarchical Combination

Serdar Erbatur[5], Deepak Kapur[1,⋆], Andrew M. Marshall[2,⋆⋆], Paliath Narendran[3,⋆⋆⋆], and Christophe Ringeissen[4]

[1] University of New Mexico (USA)
[2] Naval Research Laboratory (USA)
[3] University at Albany, SUNY (USA)
[4] LORIA – INRIA Nancy-Grand Est (France)
[5] Dipartimento di Informatica Università degli Studi di Verona (Italy)

Abstract. A novel approach is described for the combination of unification algorithms for two equational theories E_1 and E_2 which share function symbols. We are able to identify a set of restrictions and a combination method such that if the restrictions are satisfied the method produces a unification algorithm for the union of non-disjoint equational theories. Furthermore, we identify a class of theories satisfying the restrictions. The critical characteristics of the class is the hierarchical organization and the shared symbols being restricted to "inner constructors".

1 Introduction

Unification (or equation solving) is a fundamental problem in science and has been studied in different forms for thousands of years. In computer science, the concept of unification was popularized by Robinson with his resolution rule of inference for proving valid formulas in first-order predicate calculus. Since then, unification has been studied very extensively in theoretical computer science and artificial intelligence. Unification is extensively used in automated reasoning systems and other tools, such as cryptographic protocol analysis tools.

Even though the unification problem over a general first-order theory is undecidable, the problem is decidable for many first-order theories and numerous algorithms have been proposed. A critical question in the search for such methods is how to obtain a unification algorithm for the combination of non-disjoint equational theories when there exists unification algorithms for the constituent theories. The problem is known to be difficult and can easily be seen to be undecidable in the general case. Therefore, previous work has focused on identifying specific conditions and methods in which the problem is decidable. We continue the investigation in this paper, building on previous combination results, [5, 11, 18] and [13]. We are able to develop a novel approach to the non-disjoint combination problem. The approach is based on a new set of restrictions and combination method such that if the restrictions are satisfied the

⋆ Partially supported by the NSF grant CNS-0905222.
⋆⋆ Supported by an ASEE postdoctoral fellowship under contract to the NRL.
⋆⋆⋆ Partially supported by the NSF grant CNS-0905286.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 249–266, 2013.

method produces an algorithm for the unification problem in the union of non-disjoint equational theories. Furthermore, we identify a set of properties on the constituent theories, E_1 and E_2, such that theories characterized by these properties satisfy the restrictions and thus the combination algorithm is valid for these theories. The main properties of this class are: a hierarchical organization of E_1 and E_2, R_1 is a left-linear, convergent rewrite system corresponding to E_1, and the shared symbols are "inner constructors" of R_1 which are not equated in E_2.

Due to space constraints we do not attempt to give a complete review of the *related work* in this active research field but rather point the reader to a non-exhaustive list of related and important works. Combination methods for theories with disjoint signatures have been investigated in [22, 21], solved in general in [19] and extended in [5, 9]. Combination methods for theories with non-disjoint signatures is an active area. Some excellent work in this area includes [11, 18, 8, 7, 3].

2 Preliminaries

We use the standard notation of equational unification [6] and term rewriting systems [4]. The set of Σ-terms, denoted by $T(\Sigma, \mathcal{X})$, is built over the signature Σ and the (countably infinite) set of variables $\mathcal{X}$. The terms $t|_p$ and $t[u]_p$ denote respectively the subterm of t at the position p, and the term t having u as subterm at position p. The symbol of t occurring at the position p (resp. the top symbol of t) is written $t(p)$ (resp. $t(\epsilon)$). A Σ-rooted term is a term whose top symbol is in Σ. The set of variables of a term t is denoted by $Var(t)$. A term is *ground* if it contains no variables. A term t is *linear* if each variable of t occurs only once in t. A Σ-substitution σ is an endomorphism of $T(\Sigma, \mathcal{X})$ denoted by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ if there are only finitely many variables $x_1, \ldots, x_n$ not mapped to themselves. We call domain of σ the set of variables $\{x_1, \ldots, x_n\}$ and range of σ the set of terms $\{t_1, \ldots, t_n\}$. Application of a substitution σ to a term t (resp. a substitution ϕ) may be written $t\sigma$ (resp. $\phi\sigma$).

An alien subterm of a $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted term t (resp. Σ_2-rooted term) is a Σ_2-rooted subterm (resp. $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted subterm) of t such that all its superterms are $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted (resp. Σ_2-rooted).

Given a first-order signature Σ, and a set E of Σ-axioms (i.e., pairs of Σ-terms, denoted by $l = r$), the *equational theory* $=_E$ is the congruence closure of E under the law of substitutivity. By a slight abuse of terminology, E will be often called an equational theory. An axiom $l = r$ is *variable-preserving* if $Var(l) = Var(r)$. An axiom $l = r$ is *linear* (resp. *collapse-free*) if l and r are linear (resp. non-variable terms). An equational theory is *variable-preserving* (resp. linear/collapse-free) if all its axioms are variable-preserving (resp. linear/collapse-free). An equational theory E is *finite* if for each term t, there are finitely many terms s such that $t =_E s$.

A Σ-equation is a pair of Σ-terms denoted by $s =^? t$. An E-unification problem is a set of Σ-equations, $\mathcal{S} = \{s_1 =^? t_1, \ldots, s_m =^? t_m\}$. The set of variables

of $\mathcal{S}$ is denoted by $Var(\mathcal{S})$. Given a signature $\Sigma' \subseteq \Sigma$, $\mathcal{S}|_{\Sigma'}$ denotes the set of Σ'-equations occurring in $\mathcal{S}$.

A solution to $\mathcal{S}$, called an *E-unifier*, is a substitution σ such that $s_i\sigma =_E t_i\sigma$ for all $1 \leq i \leq m$. A substitution σ is *more general modulo* E than θ on a set of variables V, denoted as $\sigma \leq_E^V \theta$, if there is a substitution τ such that $x\sigma\tau =_E x\theta$ for all $x \in V$. Two substitutions θ_1 and θ_2 are *equivalent modulo* E on a set of variables V, denoted as $\theta_1 \equiv_E^V \theta_2$, if and only if $x\theta_1 =_E x\theta_2$ for all $x \in V$. A *Complete Set of E-Unifiers* of $\mathcal{S}$ is a set of substitutions denoted by $CSU_E(\mathcal{S})$ such that each $\sigma \in CSU_E(\mathcal{S})$ is an E-unifier of $\mathcal{S}$, and for each E-unifier θ of $\mathcal{S}$, there exists $\sigma \in CSU_E(\mathcal{S})$ such that $\sigma \leq_E^{Var(\mathcal{S})} \theta$.

A set of equations $\mathcal{S}$ is said to be in *standard form* over a signature Σ if and only if every equation in $\mathcal{S}$ is of the form $x =^? t$, where x is a variable and t is one of the following: (a) a variable different from x, (b) a constant, or (c) a term of depth 1 that contains no constants. It is not generally difficult to decompose equations of a given problem into simpler standard forms.

Definition 1. *A set of equations is said to be in* dag-solved form *(or* d-solved form*) if and only if they can be arranged as a list* $x_1 =^? t_1, \ldots, x_n =^? t_n$ *where (a) each left-hand side* x_i *is a distinct variable, and (b)* $\forall\, 1 \leq i \leq j \leq n$*:* x_i *does not occur in* t_j *([16]). Each* x_i *in this case is called a* solved variable*. A set of equations* S *are said to be in* Σ-solved form *if and only if it is in standard form and* $S|_\Sigma$ *is in dag-solved form.*

Definition 2. *A theory* E *is* subterm collapse-free *if and only if for all terms* t *it is not the case that* $t =_E u$ *where* u *is a strict subterm of* t*.*

Definition 3. *We say there exists a* cycle *in a unification problem if it contains a set of equations* $\bigcup_{i=1}^{n-1}\{x_i =^? t_i[x_{i+1}]\} \cup \{x_n =^? t_n[x_1]\}$ *where* $t_1, \ldots, t_n$ *are non-variable terms and* $x_1, \ldots, x_n$ *are distinct variables.*

The following is a well know property of subterm collapse-free theories (see [10]).

Proposition 1. *If* E *is subterm collapse-free, then any* E*-unification problem containing a cycle has no solution.*

Definition 4. *For a convergent rewrite system* R *we define a* constructor *of* R *to be a function symbol* f *which does not appear at the root on the left-hand side of any rewrite rule of* R*. We define an* inner constructor *to be a constructor* f *the satisfies the following additional restrictions:*

(i) f *does not appear on the left-hand side on any rule in* R*.*
(ii) f *does not appear as the root symbol on the right-hand side of any rule in* R*.*
(iii) there are no function symbols below f *on the right-hand side of any rule in* R*.*

We consider two equational theories E_1 and E_2 built over the signatures Σ_1 and Σ_2. Let $\Sigma_{(1,2)} = \Sigma_1 \cap \Sigma_2$. Here we will use some notations and methods already developed in the combination literature (See [5]).

The elements of Σ_i are called i-symbols. A term t is called an i-term if its root symbol is an i-symbol or t is a variable. Notice, that the shared symbols are both 1 and 2 symbols. We say that a Σ_i-term and a Σ_i-equation are i-pure.

We also use the notion of an *alien subterm*. An alien subterm of a 1-term (2-term), t, is a non-variable subterm s that is a 2-term (1-term). A decomposition algorithm can be defined that uses *variable abstraction* ([5]) to replace any alien subterm u in a term t by a fresh variable x and adds the equation $x =^? u$. By first converting a unification problem into standard form we have also applied variable abstraction on the entire unification problem. Decomposition problems usually employ a splitting procedure to split non-pure equations, $s =^? t$ into two pure equations $x =^? t$ and $x =^? s$ where x is a new variable. An equation between two variables is always 1 and 2-pure.

We call a term (e.g., a variable) *fresh* if it is created by applying an inference rule (or a unification algorithm) and did not previously exist.

3 Combination Procedure

We want to investigate conditions to build an $E_1 \cup E_2$-unification algorithm by using two algorithms A_1 and A_2 solving two different kinds of $E_1 \cup E_2$-unification problems. Consider an $E_1 \cup E_2$-unification problem $\mathcal{P}$ in standard form. The approach taken is as follows:

1. Run A_1 on the $\Sigma_1 \setminus \Sigma_{(1,2)}$-equations of $\mathcal{P}$. Let $\mathcal{P}'$ denote the resulting set of equations.
2. Run A_2 on the Σ_2-equations of $\mathcal{P}'$.
3. Collect the resulting problems that are in dag-solved forms.

As in disjoint combination [5], we perform a variable identification step before applying A_1 and A_2 in order to guess a priori all possible identifications of variables occurring in $\mathcal{P}$. Hence, we apply A_1 and A_2 not only on $\mathcal{P}$ but on all possible unification problems obtained from $\mathcal{P}$ by identifying some variables. In Section 3.2, we show why the classical variable identification introduced for disjoint combination is sufficient in our non-disjoint setting.

For this approach to work we need to place some restrictions on the theories and algorithms. These restrictions are introduced next, in Section 3.1. The combination algorithm, denoted by $\mathfrak{C}$, is formally presented in Figure 2. The correctness of $\mathfrak{C}$ is proved in Section 3.3. After introducing the combination algorithm and proving its correctness based on the following restrictions we show, in Section 4, how these restrictions can be satisfied by a class of equational theories.

3.1 Restrictions

We present a set of "restrictions" or assumptions on the theories and corresponding algorithms. Let Σ_1 and Σ_2 be finite signatures and let $\mathcal{X}$ be a countably infinite set of variables. Consider two *subterm collapse-free* equational theories

E_1 over the set of terms $T(\Sigma_1, \mathcal{X})$ and likewise E_2 over $T(\Sigma_2, \mathcal{X})$ such that $\Sigma_1 \cap \Sigma_2 = \Sigma_{(1,2)} \neq \emptyset$.

Restriction 1. *(Algorithm A_1) Let $\mathcal{P}$ be a set of $\Sigma_1 \setminus \Sigma_{(1,2)}$-equations. Algorithm A_1 computes a set of problems $\{\mathcal{Q}_k\}_{k\in K}$ such that*

$$\bigcup_{k\in K} CSU_{E_1\cup E_2}(\mathcal{Q}_k) \text{ is a } CSU_{E_1\cup E_2}(\mathcal{P}) \text{ and for each } k \in K:$$

(i) $\mathcal{Q}_k$ consists of $(\Sigma_1 \setminus \Sigma_{(1,2)})$-equations and $\Sigma_{(1,2)}$-equations.
(ii) $\mathcal{Q}_k$ is in standard form and $(\Sigma_1 \setminus \Sigma_{(1,2)})$-solved form.
(iii) No fresh variable occurring in a nonvariable $\Sigma_{(1,2)}$-term in $\mathcal{Q}_k$ can appear as solved in $\mathcal{Q}_k$.

Note that A_1 is a special type of algorithm that returns a "partial" solution to an $E_1 \cup E_2$-unification problem. A_1 is needed to solve some portion of the problem, namely the $\Sigma_1 \setminus \Sigma_{(1,2)}$-pure, but a standard E_1-unification algorithm is not sufficient, even for $\Sigma_1 \setminus \Sigma_{(1,2)}$-pure $E_1 \cup E_2$ problems. A standard E_1-unification algorithm may return an error even though a solution exists.

Example 1. Let $E_1 := \{h(a, x, y) = g(x * y),\ h(b, x, y) = g(y * x)\}$ and let E_2 be the commutative theory for $*$. Then, $h(a, a, z) =^? h(b, a, b)$ is not solvable in E_1 but is in $E_1 \cup E_2$.

Restriction 1 ensures that completeness is not lost by ensuring that a CSU for all the partial solutions is a CSU for the original problem. To explain Restriction 1(iii), let us note that A_1 may generate fresh $\Sigma_{(1,2)}$-equations, e.g. $z =^? f(x, y)$ where $f \in \Sigma_{(1,2)}$, together with some $\Sigma_1 \setminus \Sigma_{(1,2)}$-equations, e.g. $x =^? s, y =^? t$. If A_2 later generates $x =^? y$, then this may lead to the reapplication of A_1, to solve $x =^? s, x =^? t$. If x and y are from the initial set of variables, this problem can be discarded without loss of generality, since the variable identification performed initially generates another unification problem where x and y are already identified. The problem remains if x or y are fresh variables. In order to avoid it, we introduce Restriction 1(iii), where only the occurrences of fresh variables are restricted (see also Example 4).

Restriction 2. *(Algorithm A_2)*
Algorithm A_2 computes a finite complete set of 2-pure unifiers of 2-pure $E_1 \cup E_2$-unification problems.

Example 2. Consider $E_1 = \{r(f(x)) = r(g(x))\}$ and $E_2 = \{r(g(x)) = r(h(x))\}$ and the unification problem $r(f(x)) =^? r(h(x))$. The problem in standard form consists of the equations $\{x_1 =^? r(x_2),\ x_1 =^? r(x_3),\ x_2 =^? f(x),\ x_3 =^? h(x)\}$, which is already in $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved form. Thus the problem to be solved by A_2 is the 2-pure problem $\{x_1 =^? r(x_2),\ x_1 =^? r(x_3),\ x_3 =^? h(x)\}$, which admits the $E_1 \cup E_2$-unifiers $\sigma_1 = \{x_1 \mapsto r(f(x)), x_2 \mapsto f(x), x_3 \mapsto h(x)\}$ and $\sigma_2 = \{x_1 \mapsto r(h(x)), x_2 \mapsto h(x), x_3 \mapsto h(x)\}$. Notice that the solution σ_1 fails Restriction 2 because it is not 2-pure. Since we require that A_2 be able to solve

2-pure $E_1 \cup E_2$-unification problems with *only* 2-pure solutions, this theory is beyond the scope of our algorithm $\mathfrak{C}$. In Section 4 we give a family of theories that satisfy this restriction.

We restrict A_2 to compute only 2-pure substitutions. This is used to avoid possible reapplications of A_1 after A_2.

Restriction 3. *(Errors)*

(i) A $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted term cannot be $E_1 \cup E_2$-equal to a Σ_2-rooted term.
(ii) $E_1 \cup E_2$ is subterm collapse-free. Therefore, an $E_1 \cup E_2$-unification problem including a cycle has no solution.

Example 3. Consider $E_1 = \{f(x) = 0\}$ and $E_2 = \{g(x) = 0\}$. Restriction 3 is not satisfied since $f(x) =_{E_1 \cup E_2} 0$ where $f \in \Sigma_1 \setminus \Sigma_{(1,2)}$ and $0 \in \Sigma_2$. Hence, checking the unifiability of $f(x) \stackrel{?}{=}_{E_1 \cup E_2} g(x)$ is beyond the scope of our combination algorithm $\mathfrak{C}$.

The failure rules associated with Restriction 3 are given in Figure 1.

An Example Theory. We present a theory, called $\mathcal{E}_{AC}$, for which we applied our approach [13] and show that $\mathcal{E}_{AC}$ satisfies the restrictions 1, 2 and 3. The axioms of $\mathcal{E}_{AC}$ are as follows:

$$exp(exp(x, y), z) = exp(x, y \circledast z) \ (1) \quad (x \circledast y) \circledast z = x \circledast (y \circledast z) \ (3)$$
$$exp(x * y, z) = exp(x, z) * exp(y, z) \ (2) \quad x \circledast y = y \circledast x \ (4)$$

Here, $\mathcal{E}_{AC} = E_1 \cup E_2$ where E_1 consists of axioms (1) and (2) and E_2 axioms (3) and (4). Also, $\Sigma_1 = \{exp, *, \circledast\}$. $\Sigma_2 = \{\circledast\}$. In other words, $E_1 \cup E_2$ involves AC operator $\circledast$. The theory $\mathcal{E}_{AC}$ has the following AC-convergent system [13]:

$$exp(exp(x, y), z) \to exp(x, y \circledast z)$$
$$exp(x * y, z) \to exp(x, z) * exp(y, z)$$

Note that $\Sigma_1 \setminus \Sigma_{(1,2)} = \{exp, *\}$. In [13], we develop a procedure A_1 (called $\mathfrak{R}_1$ in [13]) that returns a $\{exp, *\}$-solved form. Furthermore, it is shown in [13] that A_1 terminates and that it is sound and complete. This implies that A_1 satisfies Restriction 1(i). The second part of the restriction, is satisfied by the checking of A_1 which does not have any rule as defined. $\mathcal{E}_{AC}$ satisfies also Restriction 2 and 3, where A_2 is the standard AC-unification algorithm. This is a consequence of results showing the fact that $\circledast$ is an inner constructor with respect to the AC-convergent rewrite system associated with axioms (1) and (2) above. Therefore, we get the following result.

Proposition 2. *$\mathcal{E}_{AC}$ satisfies Restriction 1, Restriction 2 and Restriction 3.*

We will use this theory to illustrate more results in the paper.

Conflict: If $s(\epsilon) \in \Sigma_1 \setminus \Sigma_{(1,2)}$ and $t(\epsilon) \in \Sigma_2$.

$$\{v =^? s, v =^? t\} \cup P \longrightarrow Fail$$

Cycle: If P contains a cycle

$$P \longrightarrow Fail$$

Fig. 1. Failure rules

Some of the steps below are non-deterministic, hence lead to computation paths. Throughout each path, the failure rules of Figure 1 are applied eagerly after each step. Let $\mathcal{V} = Var(\mathcal{P})$.

Step 0: Variable Identification
Guess a partition on $\mathcal{V}$ and set variables in each subset equal to each other. This requires adding fresh equations of type $u =^? w$, where $u, w \in \mathcal{V}$ and u and w belong to the same subset in the partition. Let us denote the enumeration of the partitions of $\mathcal{V}$ as $\pi_1, \pi_2, \ldots, \pi_m$. Once a partition is selected, the unification problem $\mathcal{P}$ is modified by adding the fresh equations between variables to the unification problem. We denote the modified problems by $\mathcal{P}^{\pi_i}$, $1 \leq i \leq m$.

Step 1: Run A_1
We apply A_1 to the $\Sigma_1 \setminus \Sigma_{(1,2)}$-equations of $\mathcal{P}^{\pi_i}$. If A_1 fails for all π_i, report failure and stop. Otherwise, we now have a modified, by A_1, set of unification problems from $\mathcal{P}^{\pi_i}$, $\mathcal{P}_1^{\pi_i}, \ldots \mathcal{P}_n^{\pi_i}$ for $n \geq 1$, such that all the $\Sigma_1 \setminus \Sigma_{(1,2)}$-equations in each $\mathcal{P}_j^{\pi_i}$, are in $(\Sigma_1 \setminus \Sigma_{(1,2)})$-solved form.

Step 2: Run A_2
Run Algorithm A_2 on the 2-pure equations of each $\mathcal{P}_j^{\pi_i}$. If A_2 equates variables in $\mathcal{V}$ not equated by π_{i_j} discard that particular $\mathcal{P}_j^{\pi_i}$. If none of the remaining $\mathcal{P}_1^{\pi_i}, \ldots \mathcal{P}_n^{\pi_i}$ exit with success, return failure. Otherwise, return success.

Fig. 2. $\mathfrak{C}$: Unification Algorithm for the combined theory $E_1 \cup E_2$

3.2 Variable Identification

We are concerned with the fact that some variables from $\mathcal{P}$ could be equated by A_2 and then cause reapplication of the inference rules of A_1. The way to avoid ping-ponging between the two unification algorithms is to guess, right at the beginning, a successful partition of the set of variables of $\mathcal{P}$ in a way that variables contained in each subset of each partition are set equal to each other. That is, we perform *variable identification* on $Var(\mathcal{P})$. For variable identification, we initially consider the set of all variables $Var(\mathcal{P})$ in the problem. However, some improvements are possible depending on syntactic properties of the equational theory considered, as is done in [13].

Let us assume that A_1 creates fresh variables that could cause the need for reapplication of the rules of A_1 during the application of algorithm A_2. We call

such variables "Ping-Pong Variables" as it implies back-and-forth mechanism between the algorithms.

Example 4. Consider again the theory $\mathcal{E}_{AC}$ and suppose that z is a fresh variable created by A_1 such that $z =^? exp(x_1, y_1)$, and $x_2 =^? y_2 \circledast z$. Now if this could happen the AC-unification algorithm (A_2) could equate z, by using the equation $x_2 =^? y_2 \circledast z$, to another variable, say x_3, such that $x_3 =^? x_4 * x_5$, resulting in the equation $exp(x_1, y_1) =^? x_4 * x_5$. Since this fresh equation would cause failure in the AC algorithm we would need to "ping-pong" back to A_1 to handle the fresh equation. The process could continue to repeat. Therefore, we ensure these variables don't occur in the following results.

Definition 5. *(Ping-Pong Variable) A variable x is a ping-pong variable if*

1. *x is a fresh variable created by A_1.*
2. *A_1 did not create an equation $x =^? y$, where y is a variable from the initial standard form problem.*
3. *x occurs in $x =^? t$ where t is a $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted term.*
4. *x occurs in a $\Sigma_{(1,2)}$-rooted term generated by A_1.*

Lemma 1. *Let $\mathcal{P}$ be a modified problem produced by $\mathfrak{C}$ in step (2) (i.e., an output of A_1) and assume that z is fresh variable created by A_1. If A_2 equates z to another variable y and that equality converts a subset of $\mathcal{P}$ into non-$(\Sigma_1 \setminus \Sigma_{(1,2)})$-solved form, then either z is a ping-pong variable or $\mathcal{P}$ is not unifiable.*

Proof. Let z be the fresh variable created by A_1 and assume that A_2 creates the equality $z = y$, where due to the equality a subset of equations, $\rho \in \mathcal{P}$, is not in dag-solved form. Now, if $\rho = \{x_1 =^? t_1, \ \ldots, \ x_m =^? t_m\}$ is not in solved form then one of the following situations must be present; (1) the left hand sides are not all distinct, (2) there exists a cycle in the equations. To create situation (1) the equality $z = y$ needs to create duplicate left hand sides. This implies that $x_i = z =^? t_i$ and $x_j = y =^? t_j$. If z is equated to another variable by A_2 it must occur in the 2-pure equations. Since z cannot be both equal to a $\Sigma_1 \setminus \Sigma_{(1,2)}$-term and a $\Sigma_{(1,2)}$-term, an error by Restriction 3, z must be contained in a $\Sigma_{(1,2)}$-rooted term. Thus, z is a ping-pong variable. Situation (2) implies there is a cycle between $\Sigma_1 \setminus \Sigma_{(1,2)}$-equations, a failure condition. □

Based on Restriction 1 of A_1 we get the following result.

Lemma 2. *Let $\mathcal{Q}$ be a $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved form, computed by A_1. There are no ping-pong variables in $\mathcal{Q}$.*

Example 5. In [13] it is shown that $\mathfrak{R}_1$ does not produce equations of the form $z =^? exp(x, y)$ or $z =^? x * y$, where z is a fresh variable occurring in a $\circledast$-rooted equation.

Algorithm A_2 could still create different subsets of equated variables from $\mathcal{V}$. Because $\mathcal{V}$ is finite we easily get the following result.

Lemma 3. *For each possible set of equalities $\mathcal{I}$ of the variables in $\mathcal{V}$ by the algorithm A_2 there exists a partition π_i on $\mathcal{V}$ that corresponds to $\mathcal{I}$ when the variables in each subset of π_i are equated.*

Therefore, we can "guess" a (or generate all the) partition(s) of $\mathcal{V}$. For each partition we can modify the original unification problem by pre-equating the variables in the subsets of the partition, thus guessing the action of A_2. Because there exists a partition for each action of A_2 on $\mathcal{V}$, including not equating any of the variables, we only have to apply A_1 once. This is because if for any particular partition, say π_i, algorithm A_2 equates two variables in $\mathcal{V}$ not equated by π_i we can discard that particular modified unification problem because we know there exists a partition that does correctly correspond to the action of A_2.

3.3 Termination, Soundness and Completeness

Given two theories E_1 and E_2, with corresponding algorithms A_1 and A_2, satisfying Restrictions 1 through 3, we show $\mathfrak{C}$ is terminating, sound and complete.

Theorem 1. $\mathfrak{C}$ *terminates.*

Proof. We assume that the algorithms A_1 and A_2 terminate. There is a finite number of partitions of $\mathcal{V}$, thus only a finite number of $\mathcal{P}^{\pi_i}$. Since A_1 terminates and is finitary, there are only a finite number of $\mathcal{P}^{\pi_i}_j$ problems produced. □

Lemma 4. *Let $\mathcal{P}$ be an $E_1 \cup E_2$-unification problem and σ an $E_1 \cup E_2$-unifier of $\mathcal{P}$. Then, there exists a partition π_i of $\mathcal{V}$, and an index j, such that σ is an $E_1 \cup E_2$-unifier of $\mathcal{P}^{\pi_i}_j$.*

Proof. σ naturally induces a partition of the set of variables from $\mathcal{P}$, i.e., if $x, y \in \mathcal{V}$ and $x\sigma =_{E_1 \cup E_2} y\sigma$ then x and y are in the same subset of the partition. Denote this partition as π_i. Then we can modify $\mathcal{P}$ to $\mathcal{P}^{\pi_i}$ and σ remains a unifier of $\mathcal{P}^{\pi_i}$. □

We want to prove that if a problem is unifiable there exists at least one partial solved form produced from A_1 such that if $\mathfrak{C}$ returns success after running A_2 that solved form is still intact.

Lemma 5. *Let θ be an $E_1 \cup E_2$-unifier of an $E_1 \cup E_2$-unification problem $\mathcal{P}$. There exist a partition π_i, a $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved form $\mathcal{P}'$ obtained by applying A_1 on $\mathcal{P}^{\pi_i}$, a Σ_2-solved form $\mathcal{P}''$ obtained by applying A_2 on $\mathcal{P}'_{|\Sigma_2}$ such that $\mathcal{P}'_{|\Sigma_1 \setminus \Sigma_{(1,2)}} \cup \mathcal{P}''$ is in a dag-solved form for which the related substitution σ satisfies $\sigma \leq^{Var(\mathcal{P})}_{E_1 \cup E_2} \theta$.*

Proof. By assumption, A_1 and A_2 transform an $E_1 \cup E_2$-unification problem into an equivalent set of $E_1 \cup E_2$-unification problems. Moreover, the failure rules are also sound and complete. It remains to show that the final problems (different from $Fail$) are indeed dag-solved forms. The algorithm A_2 will apply to $\mathcal{P}'_{|\Sigma_2}$

and may possibly add to $\mathcal{P}'_{|\Sigma_1 \setminus \Sigma_{(1,2)}}$ fresh equations between variables. But these additional equations cannot convert $\mathcal{P}'_{|\Sigma_1 \setminus \Sigma_{(1,2)}}$ into a non-solved form according to Lemma 1 and Lemma 2. Therefore, all final problems $\mathcal{P}'''$ different from $Fail$ are such that $\mathcal{P}'''_{|\Sigma_1 \setminus \Sigma_{(1,2)}}$ is in solved form and $\mathcal{P}'''_{|\Sigma_2}$ is in solved form. Such a final problem $\mathcal{P}'''$ is necessarily in dag-solved form, for otherwise a failure rule would apply. □

Theorem 2. *Let A_1 and A_2 be algorithms satisfying Restrictions 1 through 3. The algorithm $\mathfrak{C}$ is sound and complete, which means that $E_1 \cup E_2$-unification is finitary.*

Proof. The soundness directly follows from the fact that each time an $E_1 \cup E_2$-unification problem $\mathcal{P}$ is transformed into $\mathcal{P}'$ by $\mathfrak{C}$, then any $E_1 \cup E_2$-unifier of $\mathcal{P}'$ is an $E_1 \cup E_2$-unifier of $\mathcal{P}$. The completeness is proved by Lemma 5. □

4 A Class of Hierarchically Combinable Theories

In this section we consider a class of hierarchically defined theories. We investigate the problem of satisfying Restrictions 1 through Restriction 3. The class of theories is defined by the following set of properties. The two key properties being the hierarchical organization and the shared symbols being inner constructors. We assume that E_1 *and* E_2 *are subterm collapse-free.* The class is then defined by the following properties.

1. Properties of E_1:
   R_1 is a left-linear, convergent term rewrite system corresponding to E_1.
2. Properties of E_2:
   E_2 is a linear, finite equational theory.
3. Properties of the shared symbols:
   If $f \in \Sigma_{(1,2)}$, then f is an *inner constructor* of R_1.
   If f and g are inner constructors of R_1, then f-rooted terms cannot be equated to g-rooted terms in E_2.

The $\mathcal{E}_{AC}$ theory, is an example of a theory that is contained in the above defined class. The convergent rewrite theory associated with the first two axioms is left-linear and $\circledast$ is an inner constructor. The lower theory is the AC-theory for $\circledast$.

By assuming these properties, we can show that an E_2-unification algorithm is sufficient to satisfy Restriction 2. Moreover, these properties are sufficient to satisfy Restriction 3. With respect to Restriction 1, it is possible to build a complete "syntactic" method to generate the partial solved forms of A_1.

4.1 Satisfying Restriction 1

The first and critical step to satisfying Restriction 1 is the construction of an inference system $\mathcal{G}$ such that the standard forms of leaves in the search tree

generated by $\mathcal{G}$ are the $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved forms corresponding to a complete set of $E_1 \cup E_2$-unifiers of the input problem, thus satisfying the majority of Restriction 1. Another way to view the inference system $\mathcal{G}$ is as a method for reducing $E_1 \cup E_2$-unification problems to a set of E_2-unification problems.

We still have to identify examples of theories for which $\mathcal{G}$ can be turned into a terminating algorithm. Moreover, the computed $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved forms must satisfy Restriction 1 (iii) to avoid ping-pong variables when running $\mathfrak{C}$. Before presenting the procedure we need some results on the relation between R_1 and E_2. Immediate from the properties we have the following.

Lemma 6. *Let s_1, s_2 and t be terms such that $s_1 \longleftrightarrow_{E_2} s_2 \longrightarrow_{R_1} t$. Then there exists a term t' such that $s_1 \longrightarrow_{R_1} t' \longleftrightarrow^*_{E_2} t$.*

Lemma 7. *$\longrightarrow_{R_1} \circ \longleftrightarrow^*_{E_2}$ is terminating if and only if $\longrightarrow_{R_1}$ is terminating.*

Proof. The "only if" part is trivial. The "if" part is a consequence of the fact that $(\longrightarrow_{R_1} \circ \longleftrightarrow^*_{E_2})^* \subseteq \longrightarrow^*_{R_1} \circ \longleftrightarrow^*_{E_2}$ according to Lemma 6. □

Lemma 8. *R_1 is convergent modulo E_2. That is, for any s, t*

$$s =_{R_1 \cup E_2} t \text{ if and only if } s\downarrow_{R_1} =_{E_2} t\downarrow_{R_1}$$

Proof. The result is a consequence of [15], Theorem 3.5. There a set of restrictions is given on a rewrite theory and equational theory. The key restrictions of [15] are:
(1) R_1 be left-linear. (2) Critical pairs of R_1 are joinable modulo E_2. (3) Critical pairs of R_1 and E_2 are joinable modulo E_2. (4) $\longrightarrow_{R_1} \circ \longleftrightarrow^*_{E_2}$ is Noetherian.

(1) is satisfied by assumption. (2) is a consequence of R_1 being a convergent system. (3) is a consequence of the fact that the only shared symbols are inner-constructors of R_1. Since inner-constructors don't appear on the left-hand sides of the rules in R_1, there is no overlap between the left hand sides of R_1 with either side of the identities in E_2. Finally, (4) follows from Lemma 7. □

We modify the general syntactic E-unification procedure $\mathcal{G}$, presented in [6], which consists of a set of non-deterministic inference rules. This procedure is based on the methods developed in [14, 20]. The modifications are to the rules (i) and (ii). The resulting system is presented in Figure 3. Note that in constructing this procedure one could also start with another general E-unification method such as [14, 17, 12]. The algorithm could also be constructed from scratch as in [13].

Let $e[u]$ denote an equation with subterm u, let P and S denote sets of equations. Initially P is a set of $\Sigma_1 \setminus \Sigma_{(1,2)}$-equations and S is empty. A *computation path* in the modified $\mathcal{G}$ (from now just called $\mathcal{G}$) is a sequence of rule starting from an initial system $P : \emptyset$ and ending in a system $P : S$ such that no additional rules from $\mathcal{G}$ can be applied and S is in solved form.

We start by showing that the R_1 portion of any $E_1 \cup E_2$ proof corresponds to a computation path in $\mathcal{G}$. Note, in our procedure $\mathcal{G}$, we introduce fresh variables by considering the fresh variants of rules in R_1. These variables can be seen

(i) **LP** $\{e[u]\} \cup P : S \longrightarrow \{l =^? u,\ e[r]\} \cup P : S$
Where:
- u is a non-variable $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted subterm.
- $l \to r$ is a fresh variant of a rule in R_1.
- The root symbol of l and u are identical and $l =^? u$ is not subject to another rule before being subjected to decomposition.

(ii) **Trivial** $\{s =^? s\} \cup P : S \longrightarrow P : S$

(iii) **Decomposition**

$$\{f(u_1, \ldots, u_n) =^? f(v_1, \ldots, v_n)\} \cup P : S \longrightarrow \{u_1 =^? v_1, \ldots, u_n =^? v_n\} \cup P : S$$

Where $f \in \Sigma_1 \setminus \Sigma_{(1,2)}$.

(iv) **Orient** $\{t =^? x\} \cup P : S \longrightarrow \{x =^? t\} \cup P : S$

(v) **Variable Elimination** $\{x =^? t\} \cup P : S \longrightarrow P[t/x] : S[t/x] \cup \{x = t\}$

Fig. 3. Unification procedure $\mathcal{G}$

as existentially quantified variables [16]. Hence, an equation $x =^? t$ can be removed from a solved form when x is existentially quantified, and so existentially quantified variables do not appear in the domain of unifiers.

Lemma 9. *Let $s =^? t$ be a $\Sigma_1 \setminus \Sigma_{(1,2)}$-equation. For each $E_1 \cup E_2$-unifier θ of $s =^? t$ such that θ is R_1-normalized and $s\theta, t\theta$ are ground terms, there exists a sequence of LP applications in $\mathcal{G}$ that correspond to the R_1 steps in $s\theta \to_{R_1}^* (s\theta) \downarrow_{R_1} =_{E_2} (t\theta) \downarrow_{R_1} \leftarrow_{R_1}^* t\theta$ such that θ is an $E_1 \cup E_2$-unifier of any unification problem in this sequence.*

Proof. For an R_1-normalized θ, a ground term $s\theta$ and any R_1-rewrite sequence $s\theta \to_{R_1}^* s'$ reducing $s\theta$ to the R_1-normal form s', there exists a rewrite proof $s\theta \to^* s'$ such that no rewrite step can take place at a term introduced by any substitution. This is known as a "basic" rewrite proof (see [6], Section 4). According to the assumption and Lemma 8, we have $s\theta \longrightarrow_{R_1}^* s'$, $t\theta \longrightarrow_{R_1}^* t'$, and $s' =_{E_2} t'$ where s' and t' are respectively the R_1-normal forms of $s\theta$ and $t\theta$. We examine $s\theta \longrightarrow_{R_1}^* s'$, as $t\theta \longrightarrow_{R_1}^* t'$ follows analogously. If $s\theta \longrightarrow_{R_1}^* s'$ is a basic rewrite proof then the result follows from a simple induction argument on the length of the derivation. If there are no rewrite steps, $s\theta = s'$, then no LP step is required as it is already in normal form and E_2-unifiable. Otherwise, we get a set of rewrite steps of the form

$$s\theta = s_1 \xrightarrow[l_1 \to r_1]{p_1, \theta} s_1[r_1\theta]_{p_1} = s_2 \to \ldots \to s_n \xrightarrow[l_n \to r_n]{p_n, \theta} s_n[r_n\theta]_{p_n} = s'$$

Since at each reduction the redex is not contained in θ, each reduction corresponds to an application of LP at the same position. That is, for any $(k)^{th}$ step of the rewrite proof $s_k \xrightarrow[l_k \to r_k]{p_k, \theta} s_k[r_k\theta]_{p_k} = s_{k+1}$ there exists a LP application such that $\{s''[u]_{p_k} =^? t\} \cup P; S \Longrightarrow_{LP} \{l_k =^? u,\ s''[r_k]_{p_k} =^? t\} \cup P; S$ where

$s''\theta = s_k$. Furthermore since $u\theta = s_k|_{p_k} = l_k\theta$ and $(s''[u]_{p_k})\theta \rightarrow_{R_1} (s''[r_k]_{p_k})\theta$, θ is a $E_1 \cup E_2$-unifier of $\{l_k =^? u,\ s''[r_k]_{p_k} =^? t\}$. □

Let us denote a system $P; S$ resulting from an exhaustive application of the rules in $\mathcal{G}$ as a *$\mathcal{G}$-normal form.*

Lemma 10. *Let $\mathcal{P}$ be a set of $\Sigma_1 \setminus \Sigma_{(1,2)}$-equations. If θ is an $E_1 \cup E_2$-unifier of $\mathcal{P}$, then there exists a sequence, $\mathcal{P}; \emptyset \Longrightarrow^* \mathcal{P}'; S$, in $\mathcal{G}$ such that $\mathcal{P}'; S$ is in normal form w.r.t $\mathcal{G}$ and θ is an $E_1 \cup E_2$-unifier of $\mathcal{P}' \cup S$.*

Proof. We can modify the same proof strategy used in [6] (Lemma 4.9) but modified due to Lemma 9 and the fact that our procedure stops before considering E_2. Let θ be a R_1-normal solution to $\mathcal{P}$. For all $u =^? v \in P$, If $u\theta =_{R_1 \cup E_2} v\theta$ then by Lemma 8 $u\theta \downarrow_{R_1} =_{E_2} v\theta \downarrow_{R_1}$. By Lemma 9, there exists a $\mathcal{P}'; S$ having the same solution, obtained by a sequence of LP applications, $\mathcal{P}; S \Longrightarrow_{LP} \mathcal{P}'; S$, such that $\mathcal{P}'; S$ is a R_1-normal form. In addition, since the LP rule doesn't add equations to S, $S = \emptyset$. Thus, S is in solved form.

Now if $u\theta = v\theta$ is in R_1-normal form. Then, only rules (ii)-(v) can be applied. In this case the problem reduces to syntactic unification. In [6] a complexity measure is defined and it is shown that the syntactic unification rules will transform the system into a system lower in the complexity measure and having the same solution. In addition, it is also shown that any equation introduced into S maintains a solved form. □

We can characterize the normal forms produced in $\mathcal{G}$ by the following result.

Lemma 11. *The standard form of any $\mathcal{G}$-normal form is a $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved form.*

Proof. Consider the equations $s =^? t$ such that $s(\epsilon), t(\epsilon) \in \mathcal{X} \cup (\Sigma_1 \setminus \Sigma_{(1,2)})$. When these equations occur in a $\mathcal{G}$-normal form, $P; S$, they occur necessarily in S. Otherwise, one of the rules of $\mathcal{G}$ could be applied contradicting the assumption that $P; S$ is a $\mathcal{G}$-normal form. A $\mathcal{G}$-normal form, $P; S$, can be converted to a $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved form by converting the system into standard form. The equations in S will clearly be in $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved form and the equations in P will be converted to $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved form due to variable abstraction. In addition the conversion maintains unifiability. □

Lemma 12. *Let $\mathcal{P}$ be a set of $\Sigma_1 \setminus \Sigma_{(1,2)}$-equations. Then, the normal forms computed by $\mathcal{G}$ are equivalent to $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved forms $\{\mathcal{Q}_k\}_{k \in K}$ such that*

$$\bigcup_{k \in K} CSU_{E_1 \cup E_2}(\mathcal{Q}_k) \text{ is a } CSU_{E_1 \cup E_2}(\mathcal{P})$$

In addition, $\mathcal{Q}_k$ consists of $(\Sigma_1 \setminus \Sigma_{(1,2)})$-equations and $\Sigma_{(1,2)}$-equations.

Proof. If $\mathcal{P}$ is $E_1 \cup E_2$-unifiable the result follows from Lemma 10 and Lemma 11. If the problem is not $E_1 \cup E_2$-unifiable then there are two possibilities. First, no $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved form exist, in which case the result follows. Second, there exist one of more $\mathcal{G}$-normal forms. But since all the conversions maintain unifiability, the resulting solved forms will not be E_2-unifiable. Lastly, since R_1 can only introduce Σ_1 symbols the last result follows. □

4.2 Satisfying Restriction 2

We prove that an E_2-unification algorithm is sufficient to solve 2-pure problems modulo $E_1 \cup E_2$, the restriction on A_2. To state this result, let us consider some technical concepts introduced initially for disjoint combination [5]. We define the following bijection between the set of $(E_1 \cup E_2)$-equivalence classes of terms and a set of variables, so-called abstraction variables. This is helpful to obtain the proof of next lemma which states that variable abstractions are not affected by rewrite steps at any position in a term.

Let $\mathcal{X}$ and $\mathcal{Y}$ be disjoint sets of variables that are countably infinite. Let $T(\Sigma_1 \cup \Sigma_2, \mathcal{X})$ be the set of $\Sigma_1 \cup \Sigma_2$-terms over $\mathcal{X}$. We define a bijection $\phi : T(\Sigma_1 \cup \Sigma_2, \mathcal{X})/_{=_{E_1 \cup E_2}} \longrightarrow \mathcal{Y}$ and denote t^ϕ the ϕ-abstraction of t defined as follows:

If $t \in \mathcal{X}$, then $t^\phi = t$. If $t = f(t_1, \ldots, t_n)$ and $f \in \Sigma_2$, then $t^\phi = f(t_1^\phi, \ldots, t_n^\phi)$. Otherwise, $t^\phi = \phi([t]_{E_1 \cup E_2})$, where $[t]_{E_1 \cup E_2}$ denotes the equivalence class of t modulo $E_1 \cup E_2$.

Note that third case applies to $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted terms and ϕ-abstractions of 2-terms are 2-pure terms.

Lemma 13. *For any terms s and t, if $s \longleftrightarrow_{E_1 \cup E_2} t$, then $s^\phi =_{E_2} t^\phi$.*

Proof. Note that if s is $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted (resp Σ_2-rooted), then $s \longleftrightarrow_{E_1 \cup E_2} t$ implies that t is $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted (resp. Σ_2-rooted), due to the form of rules in R_1 and equalities in E_2, plus the subterm collapse-freeness of E_1 and E_2.

Let $s \longleftrightarrow_{E_1 \cup E_2} t$ and assume that the $\longleftrightarrow_{E_1 \cup E_2}$-step is applied at a position p in s. Let us first consider that s is Σ_2-rooted. There are two cases to consider:
1. p is not in an alien subterm. We can only apply $\longleftrightarrow_{E_2}$ at p according to our assumptions on E_1 and E_2. Then there exists an equation $l = r$ in E_2 and a substitution σ such that $s[l\sigma)]_p \longleftrightarrow s[r\sigma]_p = t$. Consider the ϕ-abstraction s^ϕ of s. Note that p is still a non-variable position in s^ϕ and we can find substitution σ' such that $s^\phi[l\sigma']_p \longleftrightarrow_{E_2} s^\phi[r\sigma']_p = t^\phi$. Thus we get $s^\phi \longleftrightarrow_{E_2} t^\phi$.
2. p is in an alien subterm occurring at position q. A $\longleftrightarrow_{E_1 \cup E_2}$-step is applied at p, which implies that $s|_p =_{E_1 \cup E_2} t|_p$, and so $s|_q =_{E_1 \cup E_2} t|_q$. Hence the alien subterms $s|_q$ and $t|_q$ are abstracted to the same variable by ϕ, that is $(s|_q)^\phi = (t|_q)^\phi$. Following the definition of ϕ-abstraction, we get $s^\phi = t^\phi$.

If s is $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted, then t is $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted. Hence, $s^\phi = t^\phi$, and so $s^\phi =_{E_2} t^\phi$.

Finally, if s is a variable, then there is no t such that $s \longleftrightarrow_{E_1 \cup E_2} t$. □

Lemma 14. *E_2-unification is sound and complete for solving 2-pure $E_1 \cup E_2$-unification problems.*

Proof. Consider an $E_1 \cup E_2$-unifier σ of $s =^? t$ where s and t are 2-pure. By induction and Lemma 13, $s\sigma =_{E_1 \cup E_2} t\sigma$ implies $s\sigma^\phi =_{E_2} t\sigma^\phi$ where in fact σ is an instance of σ^ϕ. □

Hence, an E_2-unification algorithm provides an algorithm A_2 satisfying Restriction 2 since it computes a complete set of 2-pure unifiers.

4.3 Satisfying Restriction 3

We first show that the assumptions imply the first part of Restriction 3.

Lemma 15. *A $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted term cannot be $E_1 \cup E_2$-equal to a Σ_2-rooted term.*

Proof. If a $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted term is $E_1 \cup E_2$-equal to a Σ_2-rooted term, then Lemma 13 contradicts the fact that E_2 is subterm collapse-free. □

The following lemma is useful to show that $E_1 \cup E_2$ is subterm collapse-free.

Lemma 16. *For any term t and any strict subterm u of t, if $t \longrightarrow_{R_1}^{+} t'$ then $t' \neq_{E_2} u$.*

Proof Sketch. If $t \longrightarrow_{R_1}^{+} t'$, then $t' \neq u$ for any strict subterm u of t, otherwise it would contradict the fact that E_1 is subterm collapse-free. To prove that $t' \neq_{E_2} u$, let us first introduce the notion of layer defined via the bijection ϕ used to define ϕ-abstraction in Section 4.2. A *1-layer* (resp. *2-layer*) of a term s is a 1-pure (resp. 2-pure) term which is obtained from any $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted (resp. Σ_2-rooted) subterm s' of s having no direct $\Sigma_1 \setminus \Sigma_{(1,2)}$-rooted (resp. Σ_2-rooted) superterm, by replacing each alien subterm s'' of s' by the variable $\phi([s'']_{E_1 \cup E_2})$, where $[s'']_{E_1 \cup E_2}$ denotes the equivalence class of s'' modulo $E_1 \cup E_2$. We are now ready to state the following facts:

(1) According to the form of rules in R_1, we can show that t' and t have necessarily different 1-layers. Then, this can be extended to any subterm of t, since E_1 is subterm collapse-free: t' and any subterm of t have different 1-layers. Hence, t' and u have necessarily different 1-layers.

(2) Since E_2 is subterm collapse-free, E_2 is a set of equalities between non-variable 2-pure terms having the same set of variables. Therefore, $t' =_{E_2} u$ implies that t' and u have the same 1-layers.

Consequently, it is not possible to have $t' =_{E_2} u$. □

Theorem 3. *$E_1 \cup E_2$ is subterm collapse-free.*

Proof. Assume that $t =_{E_1 \cup E_2} u$ for a strict subterm u of t. We can assume that u is in R_1-normal form. By Lemma 8, we have that $t \downarrow_{R_1} =_{E_2} u$. If $t \downarrow_{R_1} = t$, then we get $t =_{E_2} u$, which contradicts the fact that E_2 is subterm collapse-free. If $t \downarrow_{R_1} \neq t$, then we get a contradiction by Lemma 16. □

4.4 Unifiability of Partial Solved Forms

Theorem 4. *$E_1 \cup E_2$-unifiability of $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved forms is decidable if general E_2-unification is decidable.*

Proof. We use the notion of unification with (linear) constant restriction [5] and the fact that E_2-unification with linear constant restriction corresponds to general E_2-unification. A $\Sigma_1 \setminus \Sigma_{(1,2)}$-solved form can be decomposed into $\mathcal{P}_1 \cup \mathcal{P}_2$

where $\mathcal{P}_1$ denotes the solved form containing only $\Sigma_1 \setminus \Sigma_{(1,2)}$-equations between a variable and a non-variable. Let σ_1 be the idempotent substitution corresponding to the solved form $\mathcal{P}_1$. This substitution defines a constant restriction: If x is in the domain of σ_1, then x must be considered as a (free) constant in E_2. If x is in the domain of σ_1 and y occurs in $x\sigma_1$, then the constant x must not occur in the solution of y in E_2. Let $<$ be a linear constant restriction satisfying the above constant restriction. We are now ready to state that $\mathcal{P}_1 \cup \mathcal{P}_2$ is $E_1 \cup E_2$-unifiable iff $\mathcal{P}_2$ is E_2-unifiable with the linear constant restriction $<$: ($\Leftarrow$) We can that show that for any E_2-unifier σ_2 of $\mathcal{P}_2$ satisfying the linear constant restriction $<$, $\sigma_1 \circ \sigma_2$ is an $E_1 \cup E_2$-unifier of $\mathcal{P}_1 \cup \mathcal{P}_2$. ($\Rightarrow$) If σ is an $E_1 \cup E_2$-unifier of $\mathcal{P}_1 \cup \mathcal{P}_2$, then σ^ϕ is an E_2-unifier of $\mathcal{P}_2$. Moreover, σ^ϕ satisfies a linear constant restriction extending the restriction given by σ_1. Otherwise, it would be possible to contradict Restriction 3. □

As a consequence, we can consider in our assumptions a general E_2-unifiability algorithm (that is, E_2-unifiability with free symbols) instead of an E_2-unification algorithm to get an $E_1 \cup E_2$-unifiability algorithm.

4.5 Inner Constructor Definitions and Undecidable Results

Suppose we weaken Definition 4, such that inner constructors can appear on the left hand side and have constants below them. The following convergent system, R_1^u, satisfying the new definition, where $+$ is the inner constructor:

$$B(x_1 * x_2, x_3 * x_4) \to f(b + a, g(x_1, x_2, x_3, x_4))$$
$$f(a + b, g(x_1, x_2, x_3, x_4)) \to B(x_1, x_3) * B(x_2, x_4)$$

Now let our second theory, E_2^u, be the commutative theory for $+$. Then $R_1^u \cup E_2^u$-unification is undecidable due to the undecidability result for the theory of Synchronous Distributivity, $B(u * v, x * y) \longrightarrow B(u, x) * B(v, y)$, presented in [2].

5 Conclusion and Perspective

Building on previous combination results [5, 11, 18], and our own earlier work [13], we are able to identify a set of restrictions and a combination method such that if the restrictions are satisfied the method produces a unification algorithm in the union of non-disjoint equational theories. Furthermore, we identify a class of theories that satisfy the restrictions and thus the combination problem for these theories is decidable. The theory $\mathcal{E}_{AC}$, which is of interested in the area of cryptographic protocol analysis, is also presented as a theory for which the method is applicable.

Another candidate theory where the approach may be applicable is a partial theory of Cipher Block Chaining, also of interest in cryptographic protocol analysis and studied in [1]. The theory has the axiom: $bc(cons(x, Y), z) = cons(h(x, z), bc(Y, h(x, z)))$ which can be oriented from left to right. Notice that the symbol h in a left to right orientation would be an inner constructor.

The symbol h could then be given some equational properties, like AC, which would be a natural extension. The AC theory for h would form the lower theory E_2. The combination method would then be applicable for obtaining an unification algorithm for this AC-cipher block chaining theory. Interestingly, the full theory studied in [1], includes the additional axiom $bc(nil, z) = nil$. Although this would violate the restriction on subterm collapse-free theories, it may still be possible to extend the combination result to this and similar cases by first "removing" the collapsing conditions by a closure method.

With respect to efficiency, it should be possible to improve the algorithm by cutting down on the number of partitions. The first example of this can be seen in [13] where only a particular subset of the variables are needed to form the partitions and thus the number of partitions is reduced. This type of improvement may be possible in many cases. More efficient methods for enumerating these types of partitions could be developed, where the fact that these partitions will correspond to equalities between variables is taken into account. Some efficiency can be obtained if the enumeration is done by enumerating the more "granular" (less variables equated) partitions first and then preceding down the lattice of partitions to partitions where more variables are equated. If a unifier is found for one particular partition, all partitions that are both comparable and lower in the lattice are unifiable but with less general unifiers. Therefore, paths in the lattice of partitions could be pruned. Finally, it may also be possible to apply a deterministic approach. Rather than guessing a priori all possible variable identifications, allow the algorithms to ping-pong but provide a measure that is reduced at each application of an algorithm.

Acknowledgments. We are grateful to Maria Paola Bonacina and the anonymous referees for their fruitful comments. A. M. Marshall gratefully acknowledges the support of an ASEE postdoctoral fellowship. S. Erbatur was supported by the Department of Computer Science of the University at Albany and then INRIA Nancy-Grand Est during a portion of this work.

References

[1] Anantharaman, S., Bouchard, C., Narendran, P., Rusinowitch, M.: Unification modulo chaining. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 70–82. Springer, Heidelberg (2012)

[2] Anantharaman, S., Erbatur, S., Lynch, C., Narendran, P., Rusinowitch, M.: Unification modulo synchronous distributivity. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 14–29. Springer, Heidelberg (2012)

[3] Baader, F., Ghilardi, S., Tinelli, C.: A new combination procedure for the word problem that generalizes fusion decidability results in modal logics. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 183–197. Springer, Heidelberg (2004)

[4] Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press, New York (1998)

[5] Baader, F., Schulz, K.U.: Unification in the union of disjoint equational theories: Combining decision procedures. Journal of Symbolic Computation 21(2), 211–243 (1996)
[6] Baader, F., Snyder, W.: Unification theory. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 445–532. Elsevier and MIT Press (2001)
[7] Baader, F., Tinelli, C.: Combining equational theories sharing non-collapse-free constructors. In: Kirchner, H. (ed.) FroCos 2000. LNCS (LNAI), vol. 1794, pp. 260–274. Springer, Heidelberg (2000)
[8] Baader, F., Tinelli, C.: Combining decision procedures for positive theories sharing constructors. In: Tison, S. (ed.) RTA 2002. LNCS, vol. 2378, pp. 352–366. Springer, Heidelberg (2002)
[9] Boudet, A.: Combining unification algorithms. Journal of Symbolic Computation 16(6), 597–626 (1993)
[10] Bürckert, H.-J., Herold, A., Schmidt-Schauß, M.: On equational theories, unification, and (un)decidability. Journal of Symbolic Computation 8(1-2), 3–49 (1989)
[11] Domenjoud, E., Klay, F., Ringeissen, C.: Combination techniques for non-disjoint equational theories. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 267–281. Springer, Heidelberg (1994)
[12] Dougherty, D.J., Johann, P.: An improved general E-unification method. Journal of Symbolic Computation 14(4), 303–320 (1992)
[13] Erbatur, S., Marshall, A.M., Kapur, D., Narendran, P.: Unification over distributive exponentiation (sub)theories. Journal of Automata, Languages and Combinatorics (JALC) 16(2-4), 109–140 (2011)
[14] Gallier, J.H., Snyder, W.: Complete sets of transformations for general E-unification. Theoretical Computer Science 67(2-3), 203–260 (1989)
[15] Huet, G.P.: Confluent reductions: Abstract properties and applications to term rewriting systems. Journal of the ACM (JACM) 27(4), 797–821 (1980)
[16] Jouannaud, J.-P., Kirchner, C.: Solving equations in abstract algebras: A rule-based survey of unification. In: Computational Logic - Essays in Honor of Alan Robinson, pp. 257–321 (1991)
[17] Morawska, B.: General E-unification with eager variable elimination and a nice cycle rule. Journal of Automated Reasoning 39, 77–106 (2007)
[18] Ringeissen, C.: Unification in a combination of equational theories with shared constants and its application to primal algebras. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 261–272. Springer, Heidelberg (1992)
[19] Schmidt-Schauß, M.: Unification in a combination of arbitrary disjoint equational theories. Journal of Symbolic Computation 8, 51–99 (1989)
[20] Snyder, W.: A Proof Theory for General Unification, Birkhauser. Progress in Computer Science and Applied Logic, vol. 11 (1991)
[21] Tidén, E.: Unification in combinations of collapse-free theories with disjoint sets of function symbols. In: Siekmann, J.H. (ed.) CADE 1986. LNCS, vol. 230, pp. 431–449. Springer, Heidelberg (1986)
[22] Yelick, K.A.: Unification in combinations of collapse-free regular theories. Journal of Symbolic Computation 3(1-2), 153–181 (1987)

PRocH: Proof Reconstruction for HOL Light

Cezary Kaliszyk[1] and Josef Urban[2]

[1] University of Innsbruck, Austria
[2] Radboud University Nijmegen

Abstract. PRocH[1] is a proof reconstruction tool that imports in HOL Light proofs produced by ATPs on the recently developed translation of HOL Light and Flyspeck problems to ATP formats. PRocH combines several reconstruction methods in parallel, but the core improvement over previous methods is obtained by re-playing in the HOL logic the detailed inference steps recorded in the ATP (TPTP) proofs, using several internal HOL Light inference methods. These methods range from fast variable matching and more involved rewriting, to full first-order theorem proving using the `MESON` tactic. The system is described and its performance is evaluated here on a large set of Flyspeck problems.

1 Introduction, Motivation, and Related Work

Independent verification of proofs found by Automated Theorem Provers (ATPs) is not an uncommon topic in automated reasoning research. Systems like IVY [3] rely on very detailed proof output from ATPs (Otter, Prover9), which is then independently replayed and checked by a trusted system (ACL2). This technique has been used several times, e.g., for replaying the `MESON` and Otter/Prover9 proofs in HOL Light, replaying the Otter/Prover9 proofs in Mizar, and replaying the Metis proofs in HOL and Isabelle. The GDV [6] tool is even parametric: any ATP system understanding TPTP can be used for independent verification. The Metis/Isabelle combination has been used also as a part of the Sledgehammer [4] tool that uses arbitrary ATPs to discharge Isabelle proof obligations. If an ATP proof is found, and Metis can reconstruct the proof $NeededLemmas \vdash Conjecture$, then `metis(NeededLemmas)` is a valid Isabelle tactic, that is in practice used as an ATP proof importer. In HOL Light, `MESON` can be used in the same way for importing the proofs found by ATPs on FOL problems produced by the recently developed "HOL(y)Hammer" (HH) tool [2].

However, Metis and `MESON` are on average weaker than state-of-the art ATPs like Vampire and E. As ATPs and premise-selection tools get stronger, the ability of Metis and `MESON` to reconstruct (in short time usable in large ITP libraries) the ATP proof just from the proof premises decreases. Also, proofs in ITP systems like Isabelle, HOL Light and Mizar should (eventually) strive for human readability. Even if the strength of Metis and `MESON` grew, a single call to them would get hard to understand, requiring further explanation. These reasons motivate our work on a general tool that reconstructs TPTP proofs in HOL Light, using not just the proof premises, but also the steps recorded in the TPTP proof format.

[1] Proch (pronounced as "prokh") means "dust/powder" in Polish. The proof is (also) reconstructed from fine-grained inference dust.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 267–274, 2013.

2 Using Existing Approaches on HOL Light Problems

In total, the experiments with ATP-proving of HOL Light and Flyspeck theorems described in [2] have produced 7247 proofs when using Vampire, E (run under the Epar [8] scheduler) and Z3, sometimes with high timelimits (900s). Only Vampire and E produce full TPTP proofs, but Z3 also prints the necessary premises (unsat core). These proofs are pseudo/cross-minimized, i.e., each proof was re-run by all ATPs using the proof premises only, while the number of proof premises was decreasing. Using the resulting sets of premises, Epar can find 6318 proofs in 30s. This set is used for further evaluations here. Table 1 shows the performance of the potential proof-importing tools, i.e., MESON, Metis, and Prover9 run with 300s time limit. Note that (unlike MESON) both Metis and Prover9 are just run externally, i.e., not reconstructing a valid HOL proof. As mentioned above, low proof times are important for working with large ITP libraries containing (tens of) thousands of theorems, each typically proved using several of the low-level (MESON, Metis, Mizar "by", etc.) "atomic" calls. Table 2 therefore shows the performance of the above methods when using only 1 second for reconstruction.

Table 1. MESON, Metis, and Prover9 with 300s on the 6318 Epar proofs

method	MESON	Metis	Prover9
replayed	5255	4595	4672
replayed (%)	83.1	72.7	73.9

Table 2. MESON, Metis, and Prover9 with 1s on the 6318 Epar proofs

method	MESON	Metis	Prover9
replayed	5014	2803	4111
replayed (%)	79.3	44.3	65.0

Particularly the numbers obtained for Metis are considerably worse than the numbers obtained so far with Metis-based proof reconstruction in Sledgehammer [1], where only 10% of ATP proofs are lost by Metis. One possible reason is that Metis has been well-integrated with Sledgehammer, e.g., by using customized Sledgehammer-generated term orderings. Another part of explanation could be that the proofs found by HH on Flyspeck are on average harder than the proofs found by Sledgehammer on the Judgement Day benchmark. The reasons can be that the Judgement Day benchmark consists of goals that are on average easier, the HH premise selection might be more precise (allowing more involved ATP proofs), and also ATP systems like Vampire and E have been strengthened since the time of the Judgement Day evaluation.

3 PRocH System Description

The HH tool runs in parallel several (now 14) AI/ATP combinations on a given HOL problem, and if a proof is found, it is pseudo/cross-minimized by further parallel running of the ATPs (and their strategies). PRocH then follows by trying in parallel several (old and new) proof reconstruction methods. Unlike the above methods that can only use the ATP proof premises to find their own detailed proof, the most complicated of the PRocH's methods (`hh_recon`) also tries to reconstruct in HOL Light the TPTP proofs created by the ATP systems. In this its closest relative is the `isar_proof` Sledgehammer function described in [4], from which it probably differs by complete reliance on type annotations. In some sense, PRocH's `hh_recon` is so far less ambitious than `isar_proof`, because it does not yet attempt to write a HOL proof script. This also allows to treat some constructs (e.g., higher-order application) differently from `isar_proof` during the reconstruction. PRocH's use is now similar to HOL's `MESON` tactic, i.e., a call to `hh_recon[HOLPremises]` will try to justify a given HOL conjecture by going through the following stages (described more in the following subsections):

1. *Translation to FOL:* A HOL Light problem in the form $HOLPremises \vdash HOLConjecture$ is translated to an untyped FOF TPTP problem, where part of the FOF encoding of terms are annotations encoding their HOL type.
2. *Running ATPs:* An external ATP is run on the first-order problem producing a TPTP proof.
3. *Parsing:* The untyped FOF and CNF formulas in the TPTP proof are parsed back into typed HOL terms (making use of the encoded type annotations). This part also has to handle skolemization.
4. *Replaying:* The justification structure of the TPTP proof is replayed on the parsed HOL Light terms, resulting in a valid HOL Light proof.

3.1 Translation to FOL and Producing FOL Proofs

The translation to FOL is described in [2], but we show a brief example here. The translation has to encode higher-order features like lambda abstraction, currying, quantification over function variables and their application. As a leading example, consider the following higher-order theorem `FORALL_ALL`[2]

```
∀P l. (∀x. ALL (P x) l) ⟺ ALL (λs. ∀x. P x s) l
```

saying that each `x`-image of a binary relation (predicate) `P(x,y)` contains (is true for) all elements of a list `l` iff for all elements `s` of `l` the unary predicate "`P(x,s)` `is true for all x`" is true. To express this in FOL, first the lambda function is lifted from the context (its definition is created and used as an antecedent) and the higher-order applications are made explicit as follows:

```
∀l P F. (∀s. happ F s ⟺ (∀x. happ (happ P x) s))
          ⟹ ((∀x. ALL (happ P x) l) ⟺ ALL F l)
```

[2] `http://mws.cs.ru.nl/~mptp/hol-flyspeck/trunk/lists.html#FORALL_ALL`

The formulas before and after the lambda-lifting and `happ`-introduction conversions are logically equivalent in the HOL logic,[3] and such conversions are used to change the initial HOL proof state into a form $ConvHOLPremises \vdash ConvHOLConjecture$ that will later correspond to the formulas reconstructed from the ATP proof. After these conversions, the implicit polymorphic HOL type domains (`A`,`B`) are explicitly introduced as variables and quantified over, and type annotations are added for all HOL terms using the `s` and `p` wrappers (and `happ` is shortened to `i`), resulting in the following FOF formula:

```
![A,B,L,P,F]:
 (![S]:(p(s(bool,i(s(fun(B,bool),F),s(B,S))))
   <=> ![X]:p(s(bool,i(s(fun(B,bool),i(s(fun(A,fun(B,bool)),P),s(A,X))),
                       s(B,S)))))
 =>
 (![X]:p(s(bool,all(s(fun(B,bool),i(s(fun(A,fun(B,bool)),P),s(A,X))),
                    s(list(B),L))))
   <=> p(s(bool,all(s(fun(B,bool),F),s(list(B),L))))))
```

This way the HOL problem $ConvHOLPremises \vdash ConvHOLConjecture$ is translated to an untyped FOF TPTP problem $FOLPremises \vdash FOLConjecture$, on which ATPs like E and Vampire are run, producing derivations in the TPTP format [7]. Unlike the fixed and very detailed Otter/Prover9 IVY format, the TPTP proof steps may be justified by arbitrary inference method, and thus may in theory be arbitrarily hard. In practice, for E and Vampire the (overwhelming number of) proof steps are detailed and easy to check with weak ATPs. Several interesting steps from E's proof of `FORALL_ALL` are as follows:

```
fof(2,axiom,p(s(bool,t)), file('f1', aTRUTH)).
fof(4,axiom, (~(p(s(bool,f)))<=>p(s(bool,t))), file('f1', aBOOL_CASES_AX)).
fof(5,conjecture, (![A,B,L,P,F]: ... ), file('f1', cFORALL_ALL)).
fof(6,negated_conjecture, ~(![A,B,L,P,F]: ... ),
    inference(assume_negation,[status(cth)],[5])).
...
fof(25,negated_conjecture,?[X10]:?[X11]:?[X12]:?[X13]:?[X14]: ...,
    inference(variable_rename,[status(thm)],[24])).
fof(26,negated_conjecture,![X15]:(~(p(s(bool,i(s(fun(esk3_0,bool),esk6_0)...,
    inference(skolemize,[status(esa)],[25])).
...
cnf(33,plain,(~p(s(bool,f))),inference(cn,[status(thm)],[32,theory(equality)])).
...
cnf(6393,negated_conjecture,(p(s(bool,f))),
    inference(spm,[status(thm)],[6320,5512,theory(equality)])).
cnf(6404,negated_conjecture,($false),
    inference(sr,[status(thm)],[6393,33,theory(equality)])).
cnf(6405,negated_conjecture,($false),6404,['proof']).
```

[3] For HOL speakers, e.g., the `happ` functor is just the HOL identity, i.e., we use:

```
happ_def = new_definition `(happ : ((A -> B) -> A -> B)) = I`;;
happ_conv_th = prove (`!(f:A->B) x. f x = happ f x`, ... );;
happ_conv = REWR_CONV happ_conv_th;;
```

Such TPTP proofs produced by ATPs on the type-annotated input are the starting point for the HOL proof reconstruction. This is done in two stages: reconstruction of HOL terms/formulas, and reconstruction of the justification structure (HOL proof).

3.2 Reconstructing Terms and Formulas

The TPTP proof format is first parsed into suitable ML data structures using a lexer/parser combination created with `ocamllex/ocamlyacc`. For terms/formulas, parts of the HOL Light parsing mechanisms are re-used. In particular we gradually construct the intermediate HOL Light `preterm` structure, on whose final form the HOL Light `retypeckeck` function is called to obtain a HOL term. The preterm is constructed using variable/constant constructors (`Varp`), binary applications (`Combp`), abstractions (`Absp`), and type annotations (`Typing`).

Initially, the preterm just mirrors the FOL term structure, and in several passes the HOL structure is recovered from the type annotations. The recovery process might fail if the ATPs did proof-relevant operations that break the type annotation, however, at least with the resolution/paramodulation inferences done by `E` this practically does not happen. The first step is discovery of HOL type variables in formulas. For every type annotation `s(type, term)` all variables (and skolem constants) that appear in the left argument are considered to be type variables. In the next step, quantifications over such type variables are removed (they are implicitly universal in the HOL logic). During skolemization, type variables might have become arguments to newly introduced skolem functors. Such type arguments are removed, they are implicit in the HOL logic.

After that the `s` and `p` annotations are changed into `Typing` constructors with the appropriate types, and HOL Light's `retypeckeck` is called on the transformed preterm to obtain a HOL term.

3.3 Replaying ATP Proofs in HOL Light

As mentioned above, the problem $HOLPremises \vdash HOLConjecture$ is in HOL Light first converted (packaging the conversions in `HH_TAC`) to the equivalent $ConvHOLPremises \vdash ConvHOLConjecture$ problem. This problem (proof state) is then further transformed using the HOL formulas reconstructed from the ATP proof, and using mechanisms implemented by the HOL Light subgoal package to handle the ATP proof steps. Given the topologically sorted list of proof steps, for every proof step a HOL tactic is applied, depending on the type of the step. Axioms are looked up among the HOL goal assumptions (using their name) and proved using these assumptions. The negated conjecture is introduced by transforming the goal using HOL's `REFUTE_TAC` ("R") (proof by contradiction). Skolemization steps are justified using HOL's `CHOOSE_TAC` ("C"), and for plain inference steps (SZS status THM) we gradually try three increasingly complex methods: matching (`MATCH_ACCEPT_TAC` - "m"), rewriting (`REWRITE_TAC` - "r"), and HOL's full first-order ATP (`MESON_TAC` - "1" or "2" depending on the number of premises). The final contradiction concludes the HOL proof using HOL's `ACCEPT_TAC` ("A").

For the reconstruction of the proof of FORALL_ALL, the sequence of this steps is as follows: mR1rrC1111111mC1111122222221222A.

4 Evaluation

Table 4 shows 1s evaluation of the reconstruction methods tried on the 6318 Epar proofs. The methods (tactics) are described in Table 3.[4] PRocH tries (after the HH conversion) three methods in parallel, i.e., its total CPU time can be 3s. This is not very significant for the comparison, see the 300s results of MESON and Prover9 in Table 1. The methods in Table 4 are ordered from top to bottom by a greedy covering sequence (the last column), where the next method always adds most to the previous methods. The unique number of solutions, SOTAC and Σ–SOTAC [2] are metrics that show the usefulness in the whole population.

Table 3. Names and descriptions of the tactics tried for proof reconstruction

Method	Description
PRocH	HH conversion, then parallel replay with HH_RECON, MESON, and Prover9.
MESON	Standard MESON_TAC conversion then MESON and its replay.
SIMP	SIMP_TAC: Simplification by repeated conditional contextual rewriting.
Prover9	Standard Prover9 conversion then Prover9 and its proof replay.
REWRITE	REWRITE_TAC: goal simplification by repeated unconditional rewriting.
INT_ARITH	Basic algebra and linear arithmetic over the integers.
COMPLEX_FIELD	Basic "field" facts over the complex numbers.

Table 4. Performance of reconstruction tactics run in 1s on 6318 Epar proofs

Prover	Theorem (%)	Unique	SOTAC	Σ−SOTAC	Greedy (%)
PRocH	5687 (90.0)	418	0.404	2298.50	5687 (90.0)
MESON	5014 (79.3)	118	0.367	1839.30	5862 (92.7)
SIMP	2384 (37.7)	54	0.290	692.30	5968 (94.4)
INT_ARITH	407 (6.4)	4	0.236	95.95	5972 (94.5)
REWRITE	1540 (24.3)	3	0.249	382.87	5975 (94.5)
COMPLEX_FIELD	84 (1.3)	2	0.270	22.68	5977 (94.6)
Prover9	2208 (34.9)	1	0.293	646.40	5978 (94.6)

The performance of the three submethods used by PRocH are shown in Table 5. They are again ordered by their greedy covering sequence. The HH preprocessing significantly improves the Prover9-based replay, but more important for the overall performance gain is the large number (406, i.e., 6.4% of 6318) of unique solutions

[4] We tried more tactics, but they did not find more solutions. Higher times help very little, see: http://cl-informatik.uibk.ac.at/users/cek/recon_stats.html

contributed by HH_RECON. Finally, the performance of PRocH and MESON is compared in Figure 1 depending on the count of premises in the reconstructed proof. As the number of premises goes up (ATP proofs get more involved), PRocH becomes more and more necessary.

Table 5. Performance of the three submethods used by PRocH

Prover	Theorem (%)	Unique	SOTAC	Σ−SOTAC	Greedy (%)
HH + Prover9	4737 (74.9)	253	0.412	1954.00	4737 (74.9)
HH + HH_RECON	4299 (68.0)	406	0.421	1811.50	5499 (87.0)
HH + MESON	4737 (74.9)	188	0.406	1921.50	5687 (90.0)

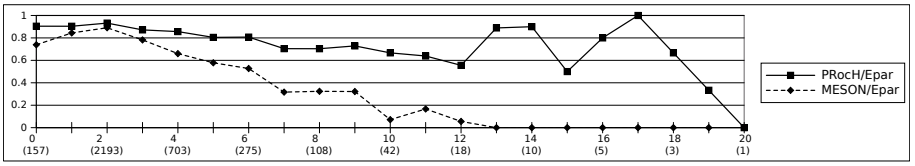


Fig. 1. PRocH and MESON dependence on premise nr. (Epar proof nr. in brackets)

5 Conclusion and Future Work

96.4% of the 6318 Epar proofs are reconstructed in 300s by some of the methods. To a great extent this validates the AI/ATP proof methods developed in [2], which were so far only verified by manual checking that the most striking (much shorter) AI/ATP proofs are really valid. 94.4% of the 6318 Epar proofs are reconstructed in 1s by one of the five (sub)methods run in parallel, i.e., the top three methods from Table 4, where PRocH consists of the three parallel submethods from Table 5. This makes the replay of more involved proofs fast and practical.

It would be good to postprocess the verbose ATP proofs into more compact, structured [10], and human-readable proofs that would be stored directly as HOL Light code. Running proof-shortening tools in a loop is a simple method that already helps a lot, e.g., when importing Otter/Prover9 proofs into Mizar. Tools for lemma and concept introduction [5,9] can be experimented with, and with stronger AI/ATP assistance are becoming more and more important.

References

1. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer, Heidelberg (2010)
2. Kaliszyk, C., Urban, J.: Learning-assisted automated reasoning with Flyspeck. CoRR, abs/1211.7012 (2012)

3. McCune, W., Matlin, O.S.: Ivy: A Preprocessor and Proof Checker for First-Order Logic. In: Computer-Aided Reasoning: ACL2 Case Studies. Advances in Formal Methods, vol. 4, pp. 265–282. Kluwer (2000)
4. Paulson, L.C., Susanto, K.W.: Source-level proof reconstruction for interactive theorem proving. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 232–245. Springer, Heidelberg (2007)
5. Pak, K.: Methods of lemma extraction in natural deduction proofs. Journal of Automated Reasoning 50, 217–228 (2013)
6. Sutcliffe, G.: Semantic derivation verification: Techniques and implementation. International Journal on Artificial Intelligence Tools 15(6), 1053–1070 (2006)
7. Sutcliffe, G., Schulz, S., Claessen, K., Van Gelder, A.: Using the TPTP language for writing derivations and finite interpretations. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 67–81. Springer, Heidelberg (2006)
8. Urban, J.: BliStr: The Blind Strategymaker. CoRR, abs/1301.2683 (2013)
9. Vyskočil, J., Stanovský, D., Urban, J.: Automated Proof Compression by Invention of New Definitions. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 447–462. Springer, Heidelberg (2010)
10. Wiedijk, F.: A synthesis of the procedural and declarative styles of interactive theorem proving. Logical Methods in Computer Science 8(1) (2012)

An Improved BDD Method for Intuitionistic Propositional Logic: BDDIntKt System Description

Rajeev Goré and Jimmy Thomson

Logic and Computation Group, Australian National University

Abstract. We previously presented a decision procedure for satisfiability and validity in propositional intuitionistic logic **Int** using Binary Decision Diagrams (BDDs). We now present some further optimisations which greatly improve performance. Primarily we focus on the impact and placement of an explicit mechanism for BDD variable ordering.

1 Introduction

We assume the reader is familiar with the syntax and Kripke semantics of propositional intuitionistic logic (**Int**). We also assume that the reader is familiar with the notions of satisfiability, validity and global logical consequence in **Int**.

For many logics, we can decide the validity of a formula φ_0 by constructing the set of all subsets of some closure $cl(\varphi_0)$, and checking whether these subsets can support a (counter) model that makes φ_0 false. If no such model exists, then we can safely declare φ_0 to be valid using this finite model property (fmp).

We previously showed how to implement this "fmp method" for **Int** without explicitly constructing all exponentially many subsets of the closure by using Binary Decision Diagrams (BDDs) [3]. Here, we present some specific optimisations that significantly improve performance of the BDD method. See: `http://users.cecs.anu.edu.au/~rpg/BDDBiKtProver/optimised-bddintkt.tgz`

1.1 A Terse Overview of the Basic Algorithm

Given φ_0, our goal is to construct a finite model $\mathcal{M} = (W_f, \preceq_f, \rho)$ by constructing a sequence of frames $(W_0, \preceq_0)$, $(W_1, \preceq_1)$, ..., $(W_f, \preceq_f)$ such that the final frame gives a model which is "canonical" in two senses: if φ_0 is satisfiable (falsifiable) then some world of W_f satisfies (falsifies) φ_0. We can then decide whether φ_0 is satisfiable or valid by checking whether such worlds exist.

We first compute $Atoms(\varphi_0) = \{\psi \in subfml(\varphi_0) \mid \psi \in Prop \vee \psi = \psi_1 \rightarrow \psi_2\}$, where $Prop$ is the set of all atomic propositions. Any subset of $Atoms(\varphi_0)$ corresponds to a classical valuation on the atoms since each atom is either in the set or not. Thus $\mathcal{W} = 2^{Atoms(\varphi_0)}$ contains all valuations over $Atoms(\varphi_0)$, and contains any valuation corresponding to any world in any model for φ_0. Under

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 275–281, 2013.

this view, each $w \in \mathcal{W}$ is both a world and a valuation. Thus the denotation of an atom $a \in Atoms(\varphi_0)$ is $[\![a]\!] := \{w \in \mathcal{W} \mid a \in w\}$. The denotations $[\![\varphi \wedge \psi]\!]$ and $[\![\varphi \vee \psi]\!]$ of arbitrary conjunctions and disjunctions from $cl(\varphi_0)$ are computed using intersections and unions of the denotations of their subformulae. Using these denotations, we compute $\preceq_{max} \subseteq \mathcal{W} \times \mathcal{W}$, an over approximation of the intuitionistic Kripke binary relation, and $\mathcal{W}_{refl}$, the subset of $\mathcal{W}$ where $\preceq_{max}$ is reflexive.

We then monotonically refine an initial approximation $W_0 \subseteq \mathcal{W}$ towards $W_f \subseteq \mathcal{W}$, using the constructed $\preceq_{max}$ relation to enforce the correct modal interpretation of the elements of $cl(\varphi_0)$ in all the worlds. Since $\preceq_i$ is just the restriction of $\preceq_{max}$ to W_i we do not need to compute it explicitly.

Once W_f has been computed, the final step is to determine which, if any, worlds in W_f satisfy and falsify φ_0, giving the satisfiability and validity of φ_0.

In [3], we started with $W_0 = \mathcal{W}$ but noted that we could instead start with $W_0 = \mathcal{W}_{refl}$. Using this latter approach, we can compute W_f as the greatest fixpoint of the sequence $W_0 := \mathcal{W}_{refl}, W_1, \cdots, W_f$ where $\overline{X} = \mathcal{W} \setminus X$ and:

$$\mathcal{W}_{refl} = \bigcap_{\phi \to \psi \in Atoms(\varphi_0)} \overline{[\![\phi \to \psi]\!]} \cup \overline{[\![\phi]\!]} \cup [\![\psi]\!]$$

$$W_i^{\varphi \to \psi} = [\![\varphi \to \psi]\!] \cup \{x \mid (x, y) \in \preceq_{max} \wedge y \in (W_i \cap [\![\varphi]\!] \cap \overline{[\![\psi]\!]})\}$$

$$W_{i+1} = W_i \cap \bigcap_{\varphi \to \psi \in Atoms(\varphi_0)} W_i^{\varphi \to \psi}$$

1.2 Basic Implementation Details

We now give some lower level details on how BDDs are actually used to perform these computations since all our optimisations are at this implementation level.

First, we analyse the initial formula φ_0 and create the set of atoms $Atoms(\varphi_0)$. During this process, we create two BDD variables a and a' for each atom, and construct the BDDs representing $[\![\psi]\!]$ and $[\![\psi]\!]'$ for each subformula ψ of φ_0 according to the definition of $[\![\cdot]\!]$. For atoms a, the BDD representing $[\![a]\!]$ is the BDD which is true exactly when the variable for a is true. For non-atomic formulae, the BDD representing $[\![\psi]\!]$ is a conjunction or disjunction of the BDDs representing the conjuncts or disjuncts as appropriate. The unprimed variables like a give rise to $\mathcal{W}$ while the primed variables like a' give rise to a "photocopy" $\mathcal{W}'$ of $\mathcal{W}$ so that $\preceq_{max}$ is actually a subset of $\mathcal{W} \times \mathcal{W}'$: see [3] for details.

During this initial creation of the BDD variables, we implicitly specify an ordering on the BDD variables according to when they are created, so if the atoms are examined in different orders then the variable ordering can be different.

After the initial analysis, we construct the BDD for $\preceq_{max}$ corresponding to the intuitionistic Kripke relation: see [3] for details. For **Int**, after defining $\mathcal{W}_{refl}$ as above, we can in fact reduce the intuitionistic relation to simply enforcing persistence, so it is a relatively simple conjunction over all atoms:

$$\preceq_{max} = \bigwedge_{a \in Atoms(\varphi_0)} [\![a]\!] \Rightarrow [\![a]\!]' = \bigcap_{a \in Atoms(\varphi_0)} (\overline{[\![a]\!]} \times \mathcal{W}') \cup (\mathcal{W} \times [\![a]\!]').$$

We next compute $\mathcal{W}_{refl}$, the initial over approximation from Section 1.1. Any global assumptions are also conjoined with this initial state. The set $\mathcal{W}_{refl}$ is a conjunction of formulae, and the conjuncts can be arbitrarily complex.

The remainder of the procedure is the fixpoint computation: while the BDD $W_i \neq W_{i-1}$, compute $W_i^{\phi \to \psi}$ for each atom $\phi \to \psi$, and conjoin the resulting BDDs with W_i to give W_{i+1}. To compute $W_i^{\phi \to \psi}$, we take W_i and replace all variables with their next-state (primed) duplicates giving W_i', and then conjoin with $[\![\phi]\!]' \wedge \neg [\![\psi]\!]'$: recall that these are (classical, not intuitionistic) operations on BDDs. Next, we conjoin with the relation BDD $\preceq_{max}$ and existentially quantify out all the primed BDD variables referring to the next state. This leaves us with a BDD representing those worlds which have some potentially good successor satisfying the required conditions. Finally, disjoin with $[\![\phi \to \psi]\!]$ to give $W_i^{\phi \to \psi}$.

Once $W_i = W_{i-1}$, all that remains is to determine whether $W_i \wedge \neg [\![\varphi_0]\!] = \bot$.

2 Optimisations

We continue to use pre-processing optimisations such as rewriting formulae to equivalent simpler formulae, and converting $\Gamma \models \psi \to \phi$ into $\Gamma, \psi \models \phi$ [3]. The following implementation rather than algorithmic changes, also helped.

Reduced ordered BDDs give unique representations of a given Boolean function and allow fast comparisons and efficient storage. The ordering placed on the BDD variables has a significant impact on the time and memory behaviour of BDD operations. Using a good ordering can give significantly smaller representations of the same Boolean function, so finding a good ordering is important.

Initial Order. When analysing the pre-processed formulae to find the atoms and precompute the BDDs for denotations, the order in allocating atoms gives an initial ordering on the BDD variables. After some experimentation, we decided on the following approach: for each non-propositional atom a, construct BDDs for all components in a depth-first left to right manner, creating new BDD variables for encountered atoms not already associated with a BDD variable. Then construct the BDD variable for a itself. Once all non-propositional atoms are handled, finally construct the BDD for the initial formula in a depth first left to right manner, stopping at atoms and creating new variables as appropriate.

Dynamic Reordering. The BDD library [1] we used has the ability to dynamically change the ordering of the BDD variables to try to find a better ordering, and this reordering takes more time when there are more and larger BDDs constructed.

Single Reordering. We investigated a number of dynamic reordering policies, changing when dynamic reordering would be attempted. Initially, we disabled all reordering and kept the ordering initially created when analysing φ_0. We then

tried a single dynamic reordering after $\mathcal{W}_{refl}$ was constructed, the idea being that the BDDs that had been constructed up to that point would be broadly similar to the ones to follow, so choosing an ordering based on that may be helpful, and doing only a single ordering would incur only minimal overheads.

More Reordering. After examining the performance on the benchmarks, we noted that there was often a bottleneck in computing $\mathcal{W}_{refl}$, before any variable ordering occurred. We therefore tried another policy of reordering more often, after every conjunct in the large conjunctions. This causes many more reorderings and thus leads to more overhead, but the improved variable ordering can result in improved efficiency, using less memory and speeding up further operations.

Automatic Reordering. We also tried a policy where reordering is performed automatic by the BDD package based on usage rather than at our behest. This has less predictable behaviour, but by reordering only when needed, it can reduce overhead while still giving the time and space benefits of more reorderings.

Why Automatic Ordering Wasn't Used Originally. When first implementing the BDD procedure, we reasoned that the large amount of memory available should be provided up-front to reduce subsequent allocation overheads. Consequently, garbage collection and dynamic reordering took a long time since they consider all BDD nodes. Also, the automatic reordering in BuDDy is triggered by an increase in the number of BDD nodes, so if there are many initial BDD nodes, automatic reorderings are far apart and slow. We therefore tried to minimise the number of reorderings. Since then, through experimenting with different options, we have found that a better option is to start with relatively few BDD nodes and attempt to keep that number small, which interacts better with reorderings.

Structure Sharing. Because of their uniqueness, BDDs can often share structure. Suppose we have 3 BDDs a, b and c, each referring to the variables in the sets VA, VB and VC, respectively, with $VA < VB < VC$ in the variable ordering. The BDD for $a \wedge b$ will share the existing BDD for b in its entirety, but cannot share the existing BDD for a since all a's "exits" to $\top$ must now point to b. Moreover, the BDD for $(a \wedge b) \wedge c$ will share the existing BDD for c in its entirety, but will share nothing with the BDD for $a \wedge b$. Constructing $a \wedge (b \wedge c)$ instead gives an identical BDD, since it represents the same Boolean function, but which contains the intermediate BDD for $b \wedge c$ in its entirety, which in turn contains the BDD for c in its entirety. This results in more sharing and greater efficiency.

Sorted Conjunctions. To try to benefit from this concept, in each large conjunction or intersection in the decision procedure above, we compute the BDD for each of the conjuncts first, then sort those BDDs such that the "lowest" BDDs (in the tree sense) are combined first, giving greater potential for sharing. In the above example, this would select c first since VC is the "lowest" set of variables, then it would conjoin with b, and finally with a, effectively computing $a \wedge (b \wedge c)$.

3 Results

We initially used the ILTP v1.1.2 propositional benchmarks [5] with a 600 second timeout. However, with some combinations of

	none-	none+	once-	once+	more-	more+	auto-	auto+	GnaA	Gna	fCube	Imogen
SYJ201	11	50	11	50	50	50	25	49	49	50	50	32
SYJ202	16	16	16	16	16	16	12	12	12	11	8	8
SYJ203	24	26	24	26	50	50	50	50	50	24	50	50
SYJ204	50	50	50	50	50	50	50	50	50	50	50	50
SYJ205	21	23	20	22	50	50	50	50	50	22	50	50
SYJ206	50	50	50	50	50	50	50	50	50	50	10	50
SYJ207	12	50	11	50	50	50	33	50	49	50	50	24
SYJ208	13	13	11	11	9	10	9	9	10	10	37	37
SYJ209	23	25	23	24	50	50	50	50	50	23	50	50
SYJ210	50	50	50	50	50	50	50	50	50	50	50	50
SYJ211	14	14	16	16	42	36	50	44	46	9	50	50
SYJ212	35	35	50	50	50	50	50	50	45	32	50	45
Total	319	402	332	415	517	512	479	514	511	381	505	496

Fig. 1. Number of instances solved by each configuration

optimisations all instances except for pigeonhole formulae were solved in less than 20 seconds each. To get a better idea of how the optimisations perform in the long term, we use an extended (unofficial) set of benchmarks (`http://www.iltp.de/download/SYJ2xx-50/SYJ2xx-50.tar.gz`) based on the ILTP, with 50 instances instead of 20 for each of the SYJ2xx formulae, except for SYJ202 and SYJ208 which go only up to 38. This unofficial set was kindly provided by the developers of the library, Jens Otten and Thomas Raths. We also compared the provers on the same randomly generated formulae as in [3]. All benchmarks were performed on 64bit Ubuntu 10.04 with a Core 2 Duo 3.0 GHz processor and 8GB RAM.

Our previous [3] results were from an implementation written in OCaml. To experiment with the BuDDy BDD library more easily and to try to control the lifetime of BDDs more precisely, we moved to a C++ implementation. In the process, some 'arbitrary' orders changed, so we include results for 'Gna' (our previous OCaml implementation), and 'GnaA' (the OCaml implementation with automatic variable reordering). All of our other results are from our C++ implementation. We tried each variant of dynamic reordering (**none**, reordering **once** after computing initial conditions, reordering **more** during each conjunction, and **auto**matic reordering) both with (+) and without (-) sorted conjunctions.

We also include results for Imogen v3.0 [4] and fCube v7.0 [2] on the extended benchmarks to compare against other state of the art theorem provers.

From Figure 1 we observe some trends. Sorting conjunctions with no or limited reordering can give significant performance gains, for example in SYJ201 and SYJ207. Without sorting conjunctions, some of these instances require over 4 million BDD nodes, indicating large BDDs and expensive operations on them. With sorted conjunctions these same instances require fewer than 8000 nodes. This effect is less pronounced when more dynamic variable ordering is applied, since the variable reordering also achieves more compact BDDs. Gna, the OCaml implementation from [3], performs significantly better than none-, the C++ implementation without any reordering or sorting, on SYJ201/207 primarily

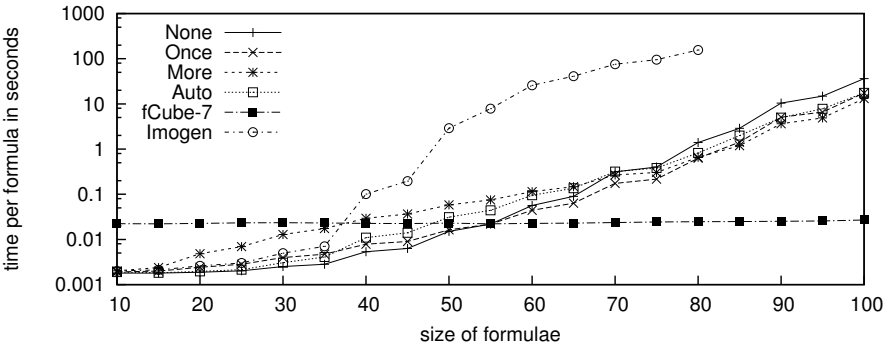


Fig. 2. Average time per random instance

because the order in which the conjunction for W_{refl} is computed is (arbitrarily) reversed, highlighting the importance of computing conjunctions efficiently.

Classes SYJ202 and SYJ208 are encodings of the pigeonhole principle, and most of the time goes into computing the BDD for the initial formula. Also, this BDD appears to be large and difficult to reduce, so reordering gives no benefit but takes up time. Reducing the effort put into dynamic variable reordering leaves more time for constructing the BDDs, and leads to better performance.

SYJ204 and SYJ210 are trivialised by normalisation and converting top level implications into global assumptions. In the case of SYJ204, once $\mathcal{W}_{refl}$ is computed the formula is immediately found to be valid, while for SYJ210, one iteration of the fixpoint is required to discover that $\mathcal{W}_f = \mathcal{W}_{refl}$.

SYJ205 and SYJ211 benefit from increased dynamic BDD variable reordering, although for SYJ205, the overhead of a single reordering on a large BDD outweighs the benefits on the remaining fixpoint computation. On both classes, the bottleneck is the initial state computation, and adding BDD variable reorderings helps to improve performance. For SYJ211, sorting conjunctions gives considerably worse results. Further inspection showed that the ILTP version of these formulae used a_i and b_i as propositions, while the extended versions simply use p_i for both. The structure is identical, but the names have been changed. We think that this interacts with our relatively naive lexicographical initial BDD variable ordering giving an"unstable" situation where, as the initial condition is computed, the "best" ordering keeps changing. If we rename propositions back to a_i and b_i as with the original problem set, then performance improves significantly. Further work in choosing an initial ordering is likely to help since, ideally, the chosen names for propositions should not have any impact on performance.

SYJ206 is rewritten to $\top$ by the formula normalisation preprocessing step. SYJ212 is not so trivialised, but dynamic variable reordering assists greatly.

On the random (typically "obviously" invalid) formulae we see that initially doing extra work does not pay off, but on larger formulae the benefits outweigh the costs. These formulae exposed a bug in fCube-7.0, again highlighting their utility. The repaired version now performs worse on some classes.

Overall, when dynamic variable ordering helps, it helps a lot. If the number of BDD nodes is small, then constructing the fixpoint formulae and dynamically reordering the BDD variables is very fast, meaning that the number of BDD nodes can probably remain small. Computing reorderings after each conjunct in the large conjunctions performed much better than we expected for this reason, although it is probably excessive. Automatic reordering based on load tends to work well for these reasons, but sometimes it can trigger a reordering at an intermediate point, and this appears to result in optimising for the wrong formulae and allows the number of BDD nodes to increase more.

4 Conclusion, Further Work and Acknowledgements

We have dramatically improved performance by paying closer attention to the BDD variable ordering. Dynamically reordering either using the automatic policy in the BDD package [1] or based on milestones in the decision procedure can keep the BDDs involved small, which translates to faster operations. A combination of the automatic reordering based on runtime usage, along with reorderings at certain milestones, may be more effective, and requires further investigation.

Variable ordering affects the size of intermediate BDDs, and computing large conjunctions by sorting them according to variable order can significantly improve performance. However, combining both dynamic variable reordering and sorting conjunctions can lead to interference and reduced performance.

The primary difference in performance between the improved version of fCube and the better performing versions of our prover is the invalid pigeon-hole class SYJ208. Since fCube is a goal-directed procedure, it can stop as soon as it finds an open branch while the BDD method must continue until it reaches a fixpoint before it can determine invalidity. Performance differs on class SYJ206 as well, but very simple pre-processing techniques trivialise this class.

The initial BDD variable ordering clearly plays a significant role in performance, even with dynamic reordering enabled. This is our current focus.

We thank the anonymous reviewers for their comments and suggestions.

References

[1] Buddy (2011), `http://sourceforge.net/projects/buddy/`
[2] Ferrari, M., Fiorentini, C., Fiorino, G.: fCube: An efficient prover for intuitionistic propositional logic. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 294–301. Springer, Heidelberg (2010)
[3] Goré, R., Thomson, J.: BDD-based automated reasoning for propositional bi-intuitionistic tense logics. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 301–315. Springer, Heidelberg (2012)
[4] McLaughlin, S., Pfenning, F.: Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 174–181. Springer, Heidelberg (2008)
[5] Raths, T., Otten, J., Kreitz, C.: The ILTP problem library for intuitionistic logic, release v1.1. Journal of Automated Reasoning (2006)

Towards Modularly Comparing Programs Using Automated Theorem Provers

Chris Hawblitzel[1], Ming Kawaguchi[2], Shuvendu K. Lahiri[1], and Henrique Rebêlo[3]

[1] Microsoft Research, Redmond, WA, USA
[2] University of California, San Diego
[3] Federal University of Pernambuco, Brazil

Abstract. In this paper, we present a general framework for modularly comparing two (imperative) programs that can leverage single-program verifiers based on automated theorem provers. We formalize (i) *mutual summaries* for comparing the summaries of two programs, and (ii) *relative termination* to describe conditions under which two programs relatively terminate. The two rules together allow for checking correctness of interprocedural transformations. We also provide a general framework for dealing with unstructured control flow (including loops) in this framework. We demonstrate the usefulness and limitations of the framework for verifying equivalence, compiler optimizations, and interprocedural transformations.

1 Introduction

The ability to compare two programs statically has applications in various domains. Comparing successive versions of a program for behavioral equivalence across various refactorings and ensuring that bug fixes and feature additions do not introduce compatibility issues, is crucial to ensure smooth upgrades [3]. Comparing different versions of a program obtained after various compiler transformations (*translation validation*) is useful to ensure that the compiler does not change the semantics of the source program [9,8]. There are two enablers for program comparison compared to the more general problem of (single) program verification. First, one of the two programs serves as an implicit specification for the other program. Second, exploiting simple and automated abstractions for similar parts of the program can lead to greater automation and scalability.

Although several systems have been developed in recent years for equivalence checking of imperative programs, there has been a lack of general framework for comparing programs. Current systems provide solutions to specific instances of the problem — translation validators focus on intraprocedural loop optimizations [8], regression verification focuses on simple interprocedural refactorings [3].

In this paper, we describe a framework for comparing programs modularly. We develop two contracts for comparing two programs: (i) First, we formalize *mutual summaries* to relate the summaries of two (possibly recursive) procedures. Mutual summaries naturally generalize postconditions used for single program

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 282–299, 2013.

$$
\begin{array}{lll}
c & \in \{\ldots, -1, 0, 1, \ldots\} & \\
\mathtt{x} & \in \mathit{Vars} & \\
R & \in \mathit{Relations} & \\
U & \in \mathit{Functions} & \\
e & \in \mathit{Expr} & ::= \mathtt{x} \mid c \mid U(e, \ldots, e) \mid \mathsf{old}(e) \mid \langle e, e \rangle \mid e.1 \mid e.2 \\
\phi & \in \mathit{Formula} & ::= \mathsf{true} \mid \mathsf{false} \mid e \ \mathsf{relop} \ e \mid \phi \wedge \phi \mid \neg\phi \mid R(e, \ldots, e) \mid \forall u.\phi \\
s & \in \mathit{Stmt} & ::= \mathsf{skip} \mid \mathsf{assert}\ \phi \mid \mathsf{assume}\ \phi \mid \mathtt{x} := e \mid \mathsf{havoc}\ \mathtt{x} \mid \\
 & & \quad s; s \mid \langle s, s \rangle \mid s \ \diamond \ s \mid s \ \bowtie \ s \mid \mathtt{x} := \mathsf{call}\ f(e, \ldots, e) \\
p & \in \mathit{Proc} & ::= \mathsf{int}\ f(\mathtt{x} : \mathsf{int}, \ldots) : \mathtt{r}\ \{\ s\ \}
\end{array}
$$

Fig. 1. A simple programming language

verification. (ii) Second, we formalize a *relative termination* specification that describes a condition $RT(f, h)$ on inputs of two procedures f and h under which the procedure h terminates whenever f terminates. Such contracts are useful to ensure that transformations do not change the terminating executions, and are important for ensuring that two transformations compose. We then provide a proof rule for checking mutual summaries and relative termination modularly. We show that these checks can be encoded using modular (single) program verifiers, and can be discharged efficiently using modern satisfiability modulo theories (SMT) solvers [2]. Finally, we provide a general framework for dealing with unstructured control flow (including loops) in this framework.

We demonstrate the usefulness of our approach on illustrative examples from equivalence checking, including conditional equivalence checking and translation validation. We encode proofs of various compiler loop optimizations such as software pipelining and loop unrolling. Our framework currently lacks the automation provided for specific forms of equivalence checking (e.g. automatically synthesizing a class of simulation relations for compiler transformations [8]). On the other hand, we show examples of comparing two programs with interprocedural changes for eliminating non-tail recursion (§4.3), monotonic behavior (§3.1), conditional equivalence (§4.3) and refactorings (§4.3) that were not amenable to automated theorem provers. We are currently incorporating the ideas in this paper into SymDiff [5], a language agnostic semantic diff framework that uses the modular program verifier Boogie [1], and the Z3 SMT solver [2].

2 Background

Figure 1 describes a programming language with recursive procedures and an assertion language. Loops and unstructured jumps can be translated into this language (§4.1). The language supports variables (*Vars*) and various operations on them. Expressions (*Expr*) can be variables, constants, or the result of applying a function U to a list of expressions. The expression $\mathsf{old}(e)$ refers to the value of e at the entry to a procedure. The expressions $e.1$ and $e.2$ extract the first and second components of a pair $\langle e_1, e_2 \rangle$. *Formula* represents Boolean valued

$$\langle \mathsf{skip}, \sigma\rangle \Downarrow \sigma \quad \langle \mathsf{assert}\ \phi, \sigma\rangle \Downarrow \sigma \quad \langle g := e, \sigma\rangle \Downarrow \mathit{eval}(e[\sigma/g]) \quad \langle \mathsf{havoc}\ g, \sigma\rangle \Downarrow \sigma'$$

$$\frac{\phi[\sigma/g]}{\langle \mathsf{assume}\ \phi, \sigma\rangle \Downarrow \sigma} \qquad \frac{\langle s_f, \sigma\rangle \Downarrow \sigma'}{\langle \mathsf{call}\ f(), \sigma\rangle \Downarrow \sigma'}\ \text{where } s_f \text{ is the body of } f \qquad \frac{\langle s_1, \sigma_1\rangle \Downarrow \sigma_1' \quad \langle s_2, \sigma_2\rangle \Downarrow \sigma_2'}{\langle\langle s_1, s_2\rangle, \langle \sigma_1, \sigma_2\rangle\rangle \Downarrow \langle \sigma_1', \sigma_2'\rangle}$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \sigma' \quad \langle s_2, \sigma'\rangle \Downarrow \sigma''}{\langle s_1; s_2, \sigma\rangle \Downarrow \sigma''} \qquad \frac{(\langle s_1, \sigma\rangle \Downarrow \sigma') \ \vee\ (\langle s_2, \sigma\rangle \Downarrow \sigma')}{\langle s_1 \ \diamond\ s_2, \sigma\rangle \Downarrow \sigma'} \qquad \frac{\langle s_1, \sigma\rangle \Downarrow \sigma' \quad \langle s_2, \sigma\rangle \Downarrow \sigma'}{\langle s_1 \ \bowtie\ s_2, \sigma\rangle \Downarrow \sigma'}$$

Fig. 2. Dynamic semantics

expressions and can be the result of relational operations on *Expr*, Boolean operations ($\{\wedge, \neg\}$), or quantified expressions ($\forall u.\phi$).

A state σ of a program at a given program location is a valuation of the variables in scope, which may include procedure parameters, locals and global variables. Figure 2 presents big-step dynamic semantics $\langle s, \sigma\rangle \Downarrow \sigma'$, which says that statement s executes to completion, transforming the initial state σ into a new state σ'. For simplicity, the formalizations in the paper (e.g. Figure 2) assume that the program contains only one variable, a global variable named g. The value of the global implicitly defines the state in such cases. (Note that g can hold tuples and arrays, which can be used to encode additional variables, procedure parameters, and procedure return values.) Most statements in *Stmt* are standard, we only describe the non-standard ones here: The assignment statement is standard (we assume an evaluation function $\mathit{eval}(e)$ that evaluates closed expressions to values). `havoc x` scrambles the value of a variable `x` to an arbitrary value. $s \ \diamond\ t$ denotes a *demonic* non-deterministic choice to either execute statements in s or t, and can be used to model conditional statements [1]. The statement $s \ \bowtie\ t$ denotes *angelic* non-deterministic choice, where the choice of executing s or t may be made in whichever way is most beneficial to the verification process. Finally, the statement $\langle s_1, s_2\rangle$ requires that the current state be a pair value ($\sigma = \langle \sigma_1, \sigma_2\rangle$); if this is satisfied, then $\langle s_1, s_2\rangle$ evaluates s_1 in state σ_1 to produce a new state σ_1', separately evaluates s_2 in state σ_2 to produce a new state σ_2', and then combines the two new states into a single new state that is a pair value: $\langle \sigma_1', \sigma_2'\rangle$. We do not expect programmers to use the statements $s \ \bowtie\ t$ and $\langle s_1, s_2\rangle$ directly; these statements are used for instrumenting programs when checking relative termination and mutual summaries respectively.

Figure 3 presents axiomatic static semantics for statements s, expressed as a weakest (liberal) precondition $\phi = \mathit{wp}(s, \phi')$. The definition of $\mathit{wp}(s, \phi')$ is standard except for the $\langle s_1, s_2\rangle$ statement and call statement. The definition of $\mathit{wp}(\langle s_1, s_2\rangle, \phi)$

$$(\mathit{wp}(s_1, \mathit{wp}(s_2, \phi[\langle g_1, g_2\rangle/g][g/g_2])[g_2/g][g/g_1])[g_1/g])[g.1/g_1, g.2/g_2]$$

$$
\begin{array}{ll}
wp(\mathsf{skip}, \phi) = \phi & wp(\mathsf{assert}\ \phi', \phi) = \phi' \wedge \phi \\
wp(\mathsf{assume}\ \phi', \phi) = \phi' \implies \phi & wp(g := e, \phi) = \phi[e/g] \\
wp(\mathsf{havoc}\ g, \phi) = \forall g.\ \phi & wp(s_1; s_2, \phi) = wp(s_1, wp(s_2, \phi)) \\
wp(s_1\ \diamond\ s_2, \phi) = wp(s_1, \phi) \wedge wp(s_2, \phi) & wp(s_1\ \bowtie\ s_2, \phi) = wp(s_1, \phi) \vee wp(s_2, \phi) \\
wp(\mathsf{call}\ f(), \phi) = \forall g'.\ R_f(g, g') \implies \phi[g'/g] &
\end{array}
$$

$$wp(\langle s_1, s_2 \rangle, \phi) = (wp(s_1, wp(s_2, \phi[\langle g_1, g_2 \rangle / g][g/g_2])[g_2/g][g/g_1])[g_1/g])[g.1/g_1, g.2/g_2]$$

Fig. 3. Static semantics

is long but not particularly deep; intuitively, it just extracts the two components of $g.1$ and $g.2$ the input state g into temporary variables g_1 and g_2, and then shuffles these values in and out of g to capture the effects of evaluating s_1 on g_1 and s_2 on g_2. Similarly, the effect of a call to a procedure f is simply replaced by an uninterpreted relation $R_f(g, g')$,

$$wp(\mathsf{call}\ f(), \phi) = \forall g'.\ R_f(g, g') \implies \phi[g'/g]$$

where g is the state before the call and g' is the state after the call completes. We often use $R = \bigcup_f \{R_f\}$ to refer to set of relation symbols over all the procedures. The following proposition connects the dynamic and static semantics:

Proposition 1. (Basic Soundness) *If $\langle s, \sigma \rangle \Downarrow \sigma'$ and $wp(s, \phi)[\mathsf{old}(g)/g]$ is valid and no symbol in R appears free in ϕ, then $\phi[\sigma/\mathsf{old}(g), \sigma'/g]$ is valid.*

In the next section, we will illustrate how the mutual summaries and the relative termination contracts constrain the relation R_f for a procedure.

3 Mutual Summaries and Relative Termination

A *program* P consists of a set of procedures $\{f_1, \ldots, f_k\}$, identified by their names. We let f, h, f_i, h_i range over procedure names. The set P contains a union of procedures from two versions of a program. We use the notation $a \doteq \lambda f, h.\ \phi(f, h)$ to be an indexed (by a pair of procedures) set of formulas such that $\phi(f, h)$ denotes the formula for the pair (f, h). We extend this notation to refer to an indexed set of expressions, constants, sets of states, *etc.*

3.1 Mutual Summaries

For any pair of procedures $f \in P$ and $h \in P$, a mutual summary $MS(f, h)$ is a relation over the input and output states of f and h. It is expressed as a formula over two copies of the input and the output variables. In general, f and h may have different sets of parameters and return values. In the case where f and h both take a single parameter x, return a single return value r, and access a single global variable g, the relation looks like:

$$\lambda x_1, x_2, g_1, g_2, r_1, r_2, g_1', g_2'.\ \phi(x_1, x_2, g_1, g_2, r_1, r_2, g_1', g_2')$$

```
int g;
void Foo1(int x){
     if (x < 100){
        g := g + x;
        Foo1(x + 1);
     }
}
```

```
void Foo2(int x){
     if (x < 100){
        g := g + 2*x;
        Foo2(x + 1);
     }
}
```

Fig. 4. Running example

where x_i, g_i, r_i and g'_i refer to the state of the parameters, input globals, return and the output globals for the i-th procedure. For brevity, we will identify a mutual summary directly with $\phi(x_1, x_2, g_1, g_2, r_1, r_2, g'_1, g'_2)$, instead of the relation (that is, avoid the λ). The semantics from Figures 2 and 3 contain just one variable g, so we write (using a curried function that accepts a pair of pre-states and a pair of post-states):

$$\lambda g_1, g_2.\lambda g'_1, g'_2.\ \phi(g_1, g_2, g'_1, g'_2)$$

Definition 1 (mutual summaries). *For procedures f and h with bodies s_f and s_h, $MS(f, h)$ holds if for every tuple of states $(\sigma_f, \sigma'_f, \sigma_h, \sigma'_h)$ such that $\langle s_f, \sigma_f\rangle \Downarrow \sigma'_f$ and $\langle s_h, \sigma_h\rangle \Downarrow \sigma'_h$, $MS(f, h)\langle\sigma_f, \sigma_h\rangle\langle\sigma'_f, \sigma'_h\rangle$ evaluates to true.*

Example 1. Consider the two programs in Figure 4. Consider the following mutual summary $MS(\texttt{Foo1}, \texttt{Foo2})$ for this pair of procedures:

$$(x_1 = x_2 \wedge x_1 \geq 0 \wedge g_1 \leq g_2) \implies (g'_1 \leq g'_2)$$

The summary states that if the procedures `Foo1` and `Foo2` are executed in two states where the respective parameters are equal and greater than 0, and if the value of the global at entry to `Foo1` is less than or equal to the value of the global at entry to `Foo2`, and if both procedures terminate, then the value of the global at exit from `Foo1` will be less than or equal to the value of the global at exist from `Foo2`.

3.2 Relative Termination

One difficulty with using mutual summaries is that they do not *compose*, since they only talk about partial correctness. Consider three procedures `A1`, `A2` and `A3` and mutual summaries $MS(\texttt{A1}, \texttt{A2})$ and $MS(\texttt{A2}, \texttt{A3})$ that express that `A1` and `A2` (respectively `A2` and `A3`) are equivalent when both procedures terminate on an input. We cannot conclude that `A1` and `A3` are equivalent when both procedures terminate on an input, since `A2` may not terminate on any input.

The relative termination specification expresses the circumstances in which f's termination implies h's termination. For any pair of procedures $f \in P$ and $h \in P$, a relative termination specification $RT(f, h)$ is a relation over the input states of f and h. It is expressed as a formula over two copies of the input variables. The relative termination specification $RT(f, h)$ is a relation

$$
\begin{array}{l}
AXIOMS = AXIOMS_{MS} \wedge AXIOMS_{RT} \\
AXIOMS_{MS} = \forall f_1, f_2, \sigma_1, \sigma_1', \sigma_2, \sigma_2'. \\
\quad R_{f_1}(\sigma_1, \sigma_1') \wedge R_{f_2}(\sigma_2, \sigma_2') \implies MS(f_1, f_2)\langle \sigma_1, \sigma_2 \rangle \langle \sigma_1', \sigma_2' \rangle \\
AXIOMS_{RT} = \forall f_1, f_2, \sigma_1, \sigma_1', \sigma_2. \\
\quad R_{f_1}(\sigma_1, \sigma_1') \wedge RT(f_1, f_2)\langle \sigma_1, \sigma_2 \rangle \implies \exists \sigma_2'.\ R_{f_2}(\sigma_2, \sigma_2')
\end{array}
$$

$$
\begin{array}{l}
CONDITIONS = \\
R \text{ does not occur free in } MS, \text{ and} \\
R \text{ does not occur free in } RT, \text{ and} \\
\forall f_1, f_2. \\
\quad (AXIOMS \implies wp(\langle s_{f_1}, s_{f_2} \rangle, MS(f_1, f_2)\ \mathsf{old}(g)\ g)[\mathsf{old}(g)/g]) \wedge \\
\quad (AXIOMS \wedge RT(f_1, f_2)\ g \implies wp(\langle s_{f_1}, at(s_{f_2}) \rangle, \mathsf{true}))
\end{array}
$$

Fig. 5. Conditions and axioms

$$\lambda x_1, x_2, g_1, g_2.\ \phi(x_1, x_2, g_1, g_2)$$

where x_i and g_i refer to the state of the parameters and input globals for the i-th procedure. In general, f and h may have different parameters. For the semantics from Figures 2 and 3, where there are no parameters, we write:

$$\lambda g_1, g_2.\ \phi(g_1, g_2)$$

Definition 2 (relative termination). *For procedures f and h with bodies s_f and s_h, $RT(f, h)$ holds if for every tuple of states (σ_f, σ_h) such that there exists a σ_f' such that $\langle s_f, \sigma_f \rangle \Downarrow \sigma_f'$ and $RT(f, h)\langle \sigma_f, \sigma_h \rangle$ is true, there is some σ_h' such that $\langle s_h, \sigma_h \rangle \Downarrow \sigma_h'$.*

Note that we do not insist that every execution from a pre-state eventually terminates, but rather at least one. For the example in Figure 4, although $RT(\texttt{Foo1}, \texttt{Foo2}) = \mathsf{true}$ is the weakest condition for relative termination (since both `Foo1` and `Foo2` always terminate), proving such a relative termination specification requires reasoning about the two programs separately using ranking functions. We later show that a stronger condition $RT(\texttt{Foo1}, \texttt{Foo2}) = x_1 \leq x_2$ can be proved modularly without any other proof rules.

3.3 Modular Checking

We now describe a method to decompose the checking that a program P satisfies a set of mutual summaries MS and relative termination specifications RT.

Figure 5 expresses the assumptions in $AXIOMS$ and the checks for guaranteeing the mutual summaries and relative termination as a condition in $CONDITIONS$ that must be satisfied. The $AXIOMS$ consists of assumptions for MS and RT specifications respectively. The $AXIOMS_{MS}$ assumes $MS(f_1, f_2)$ on the pre-post states of f_1 and f_2. The $AXIOMS_{RT}$ assumes f_2 terminates whenever it starts from a state σ_2 that is related (by $RT(f_1, f_2)$) to a terminating input state σ_1 of f_1 (we defer discussion of $at()$ until we explain $AXIOMS_{RT}$).

The check for mutual summaries is given by:

$$AXIOMS \Longrightarrow wp(\langle s_f, s_h\rangle, MS(f,h)\ \mathsf{old}(g)\ g)[\mathsf{old}(g)/g]$$

where s_f and s_h are the bodies of the procedures f and h. Intuitively, this formula prescribes a sequence of steps for checking that $MS(f,h)$ holds. First, we assume *AXIOMS* holds. Second, we assign the global state variable g an initial value of $\mathsf{old}(g)$. Third, we symbolically execute the statement $\langle s_f, s_h\rangle$, which assigns a new state to the variable g. (This has the effect of executing s_f on $g.1$ and separately executing s_g on $g.2$.) Finally, we assert that in this new state g, relative to the old state $\mathsf{old}(g)$, the mutual summary $MS(f,h)$ holds. Observe that this is analogous to modular (single) program verification, where we symbolically execute a single procedure body s_f and than assert that f's postcondition holds. The key novelty in mutual summaries is that the checking process executes *two* procedure bodies, and asserts a summary that can mention the state of both procedures.

The checking procedure ensures that if s_f and s_g execute on concrete states σ_f and σ_g, then the mutual summary $MS(f,h)$ relates the new states σ'_f and σ'_g to the old states σ_f and σ_g:

Theorem 1. *For procedures f and h with bodies s_f and s_h, if CONDITIONS is satisfied then $MS(f,h)$ holds; that is, if $\langle s_f, \sigma_f\rangle \Downarrow \sigma'_f$ and $\langle s_h, \sigma_h\rangle \Downarrow \sigma'_h$ hold for any $\sigma_f, \sigma'_f, \sigma_h, \sigma'_h$, then $MS(f,h)\langle\sigma_f, \sigma_h\rangle\langle\sigma'_f, \sigma'_h\rangle$ is true.*

The procedure bodies s_f and s_h may contain call statements. For example, the procedure bodies in Figure 4 contain recursive calls to the procedures. Suppose that procedure body s_f contains a call statement $\mathsf{call}\ f'()$ and procedure body s_h contains a call statement $\mathsf{call}\ h'()$. The weakest precondition (Figure 3) inserts an assumption $R_{f'}(g_f, g'_f)$, where g_f and g'_f are the states before and after the $\mathsf{call}\ f'()$ statement. Similarly, the weakest precondition inserts an assumption $R_{h'}(g_h, g'_h)$ for $\mathsf{call}\ h'()$. These assumption may trigger $AXIOMS_{MS}$ (Figure 5), which then produces an assumption about $MS(f,h)\langle g_f, g_h\rangle\langle g'_f, g'_h\rangle$. This assumption may be used to help prove the weakest precondition for $\langle s_f, s_h\rangle$, so that mutual summaries for recursive procedures are established inductively, assuming mutual summaries for callees while checking summaries for callers. Observe that this is analogous to modular (single) program verification, where we assume the postconditions of callees while checking the contracts in the caller.

We now show how RT are checked modularly. Figure 5 imposes the following condition for guaranteeing properties of relative termination:

$$AXIOMS \wedge RT(f,h)\ g \Longrightarrow wp(\langle s_f, at(s_h)\rangle, \mathsf{true})$$

Essentially, the formula requires that the weakest precondition of $\langle s_f, at(s_h)\rangle$ be implied by the axioms and the termination condition $RT(f,h)$. Figure 6 defines $at(s_h)$ as a transformation on s_h that include inserting an assert statement before each call in s_h, and converting each assume statement in s_h into an assertion. The purpose of checking this is to verify that all the termination assertions in

$$
\begin{array}{ll}
at(\mathsf{skip}) = \mathsf{skip} & at(s_1; s_2) = at(s_1); at(s_2) \\
at(\mathsf{assert}\ \phi) = \mathsf{assert}\ \phi & at(\langle s_1, s_2 \rangle) = \mathsf{assert\ false} \\
at(\mathsf{assume}\ \phi) = \mathsf{assert}\ \phi & at(s_1 \diamond s_2) = at(s_1) \bowtie at(s_2) \\
at(g := e) = g := e & at(s_1 \bowtie s_2) = \mathsf{assert\ false} \\
at(\mathsf{havoc}\ g) = \mathsf{havoc}\ g & at(\mathsf{call}\ f()) = \mathsf{assert}\ (\exists g'.\ R_f(g, g')); \mathsf{call}\ f()
\end{array}
$$

Fig. 6. Assertions for checking relative termination. $at(s)$ replaces a statement with a new statement.

$at(s_h)$ hold, where each termination assertion verifies that a potentially non-terminating statement actually terminates. In particular, call statements may fail to terminate and assume statements may block. If these inserted assertions are satisfied, then s_h is guaranteed to terminate:

Theorem 2. *For procedures f and h with bodies s_f and s_h, if CONDITIONS is satisfied then $RT(f, h)$ holds; that is, if $\langle s_f, \sigma_f \rangle \Downarrow \sigma'_f$ and $RT(f, h)\langle \sigma_f, \sigma_h \rangle$ is valid for any $\sigma_f, \sigma'_f, \sigma_h$, then there is some σ'_h such that $\langle s_h, \sigma_h \rangle \Downarrow \sigma'_h$.*

As with mutual summaries, relative termination is assumed for callees when checking termination of the callers, so that relative termination for recursive procedures can be established inductively.

Given these rules, one can prove that the $MS(\texttt{Foo1}, \texttt{Foo2}) = (x_1 = x_2 \wedge x_1 \geq 0 \wedge g_1 \leq g_2) \implies (g'_1 \leq g'_2)$ holds for Figure 4. One can similarly prove that $RT(\texttt{Foo1}, \texttt{Foo2}) = x_1 \leq x_2$ holds, assuming it holds for nested pairs of calls. In both cases, we use the fact that whenever $x_1 \leq x_2$, `Foo2` cannot execute the nested recursive call to itself without `Foo1` calling itself. Although the condition $RT(\texttt{Foo1}, \texttt{Foo2}) = x_1 \geq x_2$ holds, it cannot be proved modularly using only these proof rules. This is expected, as these rules are only sound, but not complete.

3.4 Proof Sketch

We have proven the main theorems (Theorem 1 and Theorem 2).[1] The key lemma is a proof that the axioms in Figure 5 are valid.

Lemma 1. (Full Soundness) *If CONDITIONS is satisfied and $\langle s, \sigma \rangle \Downarrow \sigma'$ and $(AXIOMS \implies wp(s, \phi)[\mathsf{old}(g)/g])$ is valid and no symbol in R appears free in ϕ, then $\phi[\sigma/\mathsf{old}(g), \sigma'/g]$ is valid.*

The main challenge in the proof of this lemma is that the validity of the axioms depend on the conditions in Figure 5, which in turn mention the axioms. To break this circularity, we build up the axiom validity inductively on the call depth (maximum number of nested calls) in an execution $\langle s, \sigma \rangle \Downarrow \sigma'$. The base case uses empty relations $R_{f_1} = \emptyset, \ldots, R_{f_k} = \emptyset$, meaning that there are no calls

[1] Detailed proofs are available off the extended technical report page at `http://research.microsoft.com/apps/pubs/?id=154989`.

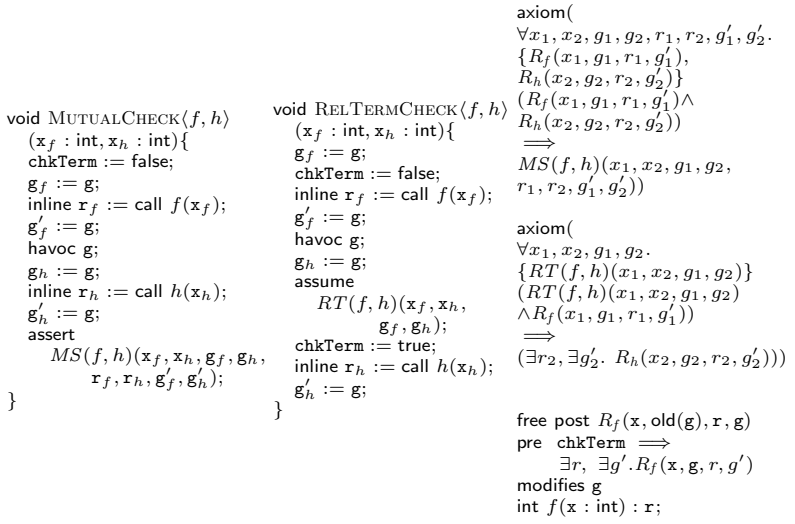


Fig. 7. Encoding the rules into a modular program verifier Boogie

(call depth 0). The inductive case assumes relations $R_{f_1}, \ldots, R_{f_k}$ for call depth n, and increases the membership of these relations to include executions with call depth $n + 1$.

3.5 Encoding in Boogie

By exploiting the close analogy between mutual summaries, relative termination, and traditional modular (single) program verification, we can use automated single-program verification tools like Boogie to check mutual summaries and relative termination for a subset of programs described in Figure 1. We restrict ourselves to the case of programs that do not contain any angelic choice statements ($s \bowtie t$), and the only use of a demonic choice ($s \diamond t$) or an assume statement in the program syntax comes from the modeling of conditional statements. These restrictions along with absence of loops ensure that the only source of non-termination comes from nested procedure calls. Figure 7 shows an encoding of the axioms and conditions from Figure 5.

First, we define the predicate R_f for each procedure f over the input and output symbols, and add it as a "free" postcondition for f. The "free" postconditions of a procedure are unchecked postconditions that are only assumed at call sites, but never asserted. The precondition (guarded by a ghost global variable `chkTerm`) captures the assertion for checking relative termination.

Second, we define a procedure MutualCheck$\langle f, h\rangle$ that checks a mutual summary $MS(f, h)$. Note that the global variable `chkTerm` is set to `false` — this has the effect of turning off the relative termination assertions at call-sites. We write "inline `r` := call f(`x`)" to inline the body of f (upto calls). The `assert` checks the mutual summary $MS(f, h)$ after executing f and h on their copies of globals.

Third, we define a procedure RelTermCheck$\langle f, h\rangle$ that defines how to check the relative termination of h with respect to f under $RT(f, h)$. The setting `chkTerm` = `true` enables assertions of termination before potentially non-terminating statements while checking the body of h.

Finally, the `axiom`(.) encode the axioms in *AXIOMS*. Each axiom has a set of *triggers* that control when the axioms are instantiated [2]. The triggers represent a list of expressions inside {.}, containing all the bound variables in a quantified axiom.

4 Applications

In this section, we show the application of our approach towards various examples of intraprocedural and interprocedural transformations.[2]

4.1 Loops and Unstructured Control

In this section, we provide a general framework for translating arbitrary unstructured control flow graphs (including loops) into recursive procedures. Unstructured control flow is fairly common when dealing with low-level programs such as binaries. The general scheme requires that certain locations in a program be decorated as special *function labels* (`FLABEL`). Given a program where every cycle passes through at least one function label, the following simple algorithm transforms this program into a set of mutually recursive procedures. First, each function label becomes a procedure, whose parameters are all local variables and procedure parameters in scope. Second, the body for each procedure is the collection of statements reachable from that procedure's function label via paths that do not pass through a function label. Finally, each goto statement to a function label (or implicit fall-through to a function label) becomes a tail-recursive call to the procedure for that function label. Notice that in the second step, the same statements might be included separately in different procedures, if those statements are reachable from different function labels. In the worst case, each statement could be included in each generated procedure, so the worst-case size of the resulting program is the product of the original program size and the number of function labels. Figure 8 shows an example of such loop extraction.

Although the general scheme allows the user flexibility in the choice of `FLABELS` to eliminate loops, one can automate the extraction for structured programs. For such programs, it suffices to identify the set of loop heads as `FLABEL`. We have implemented a variant of this scheme in Boogie to automate the translation of structured loops into tail-recursive procedures. [3] For the examples in this paper, we however explicitly mention the set of `FLABEL` locations.

[2] Detailed Boogie examples used in this paper are available off the extended technical report page at `http://research.microsoft.com/apps/pubs/?id=154989`.

[3] The exact Boogie options to be specified are ``/printInstrumented /extractLoops /deterministicExtractLoops ''.

4.2 Intraprocedural Translation Validation

This section describes the use of mutual summaries to perform (intraprocedural) translation validation [9], focusing on the validation of compiler loop optimizations. The validation consists of three steps: (1) eliminating unstructured control (including loops), (2) providing mutual summaries, (3) user-specified *inlining* of calls to recursive procedures zero or more times to express the effect of loop optimizations such as loop unrolling.

In describing the examples in this section, we follow the approach by Kundu et al. [4] to express *parameterized* versions of programs, where the effect of a loop-free and call-free block of statements is modeled as an application of an uninterpreted function. The type of the globals is an uninterpreted type `T`, and there is a single global `g` of this type representing the global state unless otherwise noted.

```
void A(){
  i := 0;
While1:
  if(i < E(n)){
     g := S1(g,i);
     g := S2(g,i);
     i := i + 1;
L1: //FLABEL
     goto While1;
  }
}

void B(){
  i := 0;
  g := S1(g,i);
While2:
  if(i < E(n)-1){
     g := S2(g,i);
     i := i + 1;
L2: //FLABEL
     g := S1(g,i);
     goto While2;
  }
  g := S2(g,i);
  i := i + 1;
}
```

(a)

```
void A'(){
  i := 0;
  if(i < E(n)){
     g := S1(g,i);
     g := S2(g,i);
     i := i + 1;
     r := call L1(i);
  }
}

int L1(int i){
  i' := i;
  if(i' < E(n)){
     g := S1(g,i');
     g := S2(g,i');
     i' := i' + 1;
     r := call L1(i');
     return r;
  }
}
```

(b)

```
void B'(){
  i := 0;
  g := S1(g,i);
  if(i < E(n)-1){
     g := S2(g,i);
     i := i + 1;
     r := call L2(i);
     return ;
  }
  g := S2(g,i);
  i := i + 1;
}

int L2(int i){
  i' := i;
  g := S1(g,i');
  if(i' < E(n)-1){
     g := S2(g,i');
     i' := i' + 1;
     i' := call L2(i');
     return i';
  }
  g := S2(g,i');
  i' := i' + 1;
  return i';
}
```

(c)

Fig. 8. Example of software pipelining. (a) Input programs and (b,c) programs after loop extraction for `A` and `B` respectively.

Software Pipelining. Figure 8 describes the encoding of software pipelining, where `E`, `S1` and `S2` represent uninterpreted predicates or functions. The optimization can be expressed as a composition of two transformations [4] (a) transformation from `A` to `B`, and (b) replacing the sequence of statements

$$\mathtt{g} := \mathtt{S2}(\mathtt{g}, \mathtt{i}); \mathtt{i} := \mathtt{i} + 1; \mathtt{g} := \mathtt{S1}(\mathtt{g}, \mathtt{i});$$

in B with

$$\mathtt{g} := \mathtt{S1}(\mathtt{g}, \mathtt{i}+1); \mathtt{g} := \mathtt{S2}(\mathtt{g}, \mathtt{i}); \mathtt{i} := \mathtt{i}+1$$

We only describe the proof of the transformation from A to B. The latter follows under the assumption that the call-free statements `g := S1(g,i+1)` and `g := S2(g,i)` commute. Although it is easy to see that the second transformation cannot affect termination, a rigorous proof of the composed transformation would need the use of relative termination (omitted for brevity).

Apart from the use of FLABEL to extract loops into recursive procedures, the interesting part of the proof is in the following mutual summaries used to express the relationships between the two versions:

- $MS(\mathtt{A}', \mathtt{B}') \doteq (E(n) > 0 \wedge g_1 = g_2) \implies g_1' = g_2'$
- $MS(\mathtt{L1}, \mathtt{L2}) \doteq (E(n) > 0 \wedge i_1 = i_2 \wedge g_1 = g_2 \wedge i_2 < E(n)) \implies (g_1' = g_2' \wedge r_1 = r_2)$
- $MS(\mathtt{L1}, \mathtt{L1}) \doteq i_1 \geq E(n) \implies (g_1' = g_1 \wedge r_1 = i_1)$

The constraint $E(n) > 0$ present in the summaries is the condition under which the transformation is sound. The constraint $i_2 < E(n)$ is a precondition for L2 that is expressed as an antecedent in the mutual summary for $MS(\mathtt{L1}, \mathtt{L2})$. Finally, the $MS(\mathtt{L1}, \mathtt{L1})$ is a postcondition for L1 that is required to reason about the last iteration of the loop in L1 — it expresses that when input $i \geq E(n)$, then L1 does not transform the state and returns the input i. We believe that only the last postcondition is the additional price paid for using mutual summaries instead of traditional simulation relations in earlier works [4].

Loop Unrolling. Figure 9 describes the example of loop unrolling, where B performs two iterations of the loop whenever `i + 1 < E(n)`. The interesting part for the proof is that the body of extracted procedure L1 has to be inlined *once* inside itself to match it up with L2. We omit the resulting mutual summaries that express $(\mathtt{A}', \mathtt{B}')$ and $(\mathtt{L1}, \mathtt{L2})$ are equivalent (modulo termination).

```
void A(){
  i := 0;
L1: //FLABEL
  if(i < E(n)){
     g := S1(g,i);
     i := i + 1;
     goto L1;
  }
}
```

```
void B(){
  i := 0;
L2; //FLABEL
  if(i + 1 < E(n)){
     g := S1(g,i); i := i + 1;
     g := S1(g,i); i := i + 1;
     goto L2;
  }
  if(i  < E(n)){
     g := S1(g,i); i := i + 1;
  }
}
```

Fig. 9. Example of loop unrolling. Input programs A and B.

```
T a, b; //globals
void A(){
  i := 0;
L1: //FLABEL
  if(i < E(n)){
     a := S1(a,i);
     if (F(b)){
       a := S2(a,i);
     }
     i := i + 1;
     goto L1;
  }
}
```

```
void B(){
  i := 0;
  if (F(b)){
L2: //FLABEL
     if(i < E(n)){
        a := S1(a,i); a := S2(a,i); i := i + 1;
        goto L2;
     }
  } else {
L3: //FLABEL
     if(i < E(n)){
        a := S1(a,i); i := i + 1;
        goto L3;
     }
  }
}
```

Fig. 10. Example of loop unswitching

Loop Unswitching. Figure 10 describes the example of loop unswitching. Here the loop in `A` (at `L1`) is split into two loops in `B` since the condition `F(b)` does not change in the loop. The mutual summaries for this proof are:

- $MS(\mathtt{A}', \mathtt{B}') \doteq (a_1 = a_2 \wedge b_1 = b_2) \implies a_1' = a_2'$
- $MS(\mathtt{L1}, \mathtt{L2}) \doteq (F(b_1) \wedge i_1 = i_2 \wedge a_1 = a_2 \wedge b_1 = b_2) \implies (a_1' = a_2' \wedge r_1 = r_2)$
- $MS(\mathtt{L1}, \mathtt{L3}) \doteq (\neg F(b_1) \wedge i_1 = i_2 \wedge a_1 = a_2 \wedge b_1 = b_2) \implies (a_1' = a_2' \wedge r_1 = r_2)$

The interesting part of the mutual summaries is the presence of the conditions under which the loop `L1` matches with `L2` or `L3`. This is also one of the instances where the mutual summaries relate one procedure (`L1`) to multiple procedures (`L2` and `L3`).

Other Compiler Optimizations. In addition to the optimizations shown in this section, we have been able to prove many other examples of loop optimizations handled by previous works [13,4]. The notable exceptions are transformations such as loop reversal and loop interchange that may change the order of updates to an array. Previous works have used a special PERMUTE rule [13], that tries to permute the order of updates in a loop. We are currently investigating encoding this rule using mutual summaries and relative termination. Nevertheless, our approach already handles many examples of interprocedural transformations that are beyond the ability the PERMUTE rule (§ 4.3).

4.3 Interprocedural Transformations

In this section, we show the applications of our approach towards instances of interprocedural transformation.

Compiler Optimizations. Our approach can be used to prove various compiler optimizations that require global (or interprocedural) analysis. The proof of *tail recursion elimination* can be done easily after the loop is extracted into a tail-recursive procedure using `FLABEL`. The proof for *inlining* will be similar

```
T a, b; //globals
void A(){
  call B(0);
}
void B(int i){
  if(i < E(n)){
     call B(i+1);
     a := S1(a,i);
     b := S2(b,i);
  }
}
```

```
void C(){
  call D(0); call E(0);
}
void D(int i){
  if(i < E(n)){
     call D(i+1); a := S1(a,i);
  }
}
void E(int i){
  if(i < E(n)){
     call E(i+1); b := S2(b,i);
  }
}
```

Fig. 11. Example of restricted interprocedural loop fission

to the proof of *loop unrolling* discussed in earlier section. Similarly, *global constant propagation* can be encoded using mutual summaries that express that a particular global or return variable has a constant value.

In addition to common compiler optimizations, Figure 11 demonstrates a transformation of a single non-tail recursive procedure into two non-tail recursive procedures (corresponds to a restricted interprocedural version of *loop fission* optimization). This can be handled using mutual summaries (omitted).

```
f1(n) {
  if (n == 0) {
    return 1;
  } else {
    return
     n * f1(n - 1);
  }
}
```

```
f2(n, a) {
  if (n == 0) {
    return a;
  } else {
    return
      f2(n - 1, a * n);
  }
}
```

Fig. 12. Example for tail vs. non-tail recursive factorial

Figure 12 shows two implementations of factorial, one tail recursive and one not tail recursive. We can prove that these compute the same result ($\texttt{f1}(n) = \texttt{f2}(n, 1)$) using the following mutual summary: $MS(\texttt{f1}, \texttt{f2}) \doteq (n_1 = n_2) \implies (r_1 * a_2 = r_2)$.

Conditional Equivalence. Bug fixes and feature additions result in two versions of a program that are behaviorally equivalent only under a subset of inputs. We show that mutual summaries can be used for showing *conditional equivalence* even for recursive procedures. Figure 13 contains two versions of a procedure `f` (denoted as `f1` and `f2` respectively) that recursively evaluates an expression rooted at the argument x. The new version differs in functionality when an additional argument `u` is provided that indicates "unsigned" arithmetic instead of the signed arithmetic represented by {+,-}. The following mutual summary $MS(\texttt{f1}, \texttt{f2})$ validates that the two procedures agree when u is off: $(x_1 = x_2 \wedge u = 0) \implies r_1 = r_2$.

Most examples in this Section have different set of inputs for the two versions, and thus not amenable to be abstracted with a common uninterpreted function [3]. Let us briefly comment on the relationship with previous works that use identical uninterpreted functions to abstract equivalent procedures [8,3]. Using an uninterpreted function (instead of mutual summaries) to represent equivalent

```
int f1(int x){
   if (Op[x] = 0)
     return Val[x];
   a := f1(A[x]);
   b := f1(B[x]);
   if (Op[x] = 1)
     return a + b;
   else if (Op[x] = 2)
     return a - b;
   else
     return 0;
}
```

```
int f2(int x, int u){
   if (Op[x] = 0)
     return Val[x];
   a := f2(A[x], u);
   b := f2(B[x], u);
   if (Op[x] = 1){
     if (u) return uAdd(a,b);
     else   return a + b;
   } else if (Op[x] = 2){
     if (u) return uSub(a,b);
     else   return a - b;
   } else   return 0;
}
```

Fig. 13. Example for feature addition and conditional equivalence

```
void D(ref x){
  d[x] := U(d[x]);
}
void A(ref x){
  if (x != null){
    call A(next[x]);
    call D(x);
  }
}
void B(ref x){
  if (x != null){
    call D(x);
    call B(next[x]);
  }
}
```

(a)

```
void AD(x,y){
  inline call A(x);
  call D(y);
}
void DA(x,y){
  call D(y);
  inline call A(x);
}
void DD(x,y){
  inline call D(x);
  inline call D(y);
}
```

(b)

Mutual summaries (A vs. B)

$MS(\mathtt{A},\mathtt{B}) : (x_1 = x_2 \wedge d_1 = d_2) \implies d_1' = d_2'$

$MS(\mathtt{A},\mathtt{A}) : (x_1 = x_2 \wedge d_1 = d_2) \implies d_1' = d_2'$

$MS(\mathtt{D},\mathtt{D}) : (x_1 = x_2 \wedge d_1 = d_2) \implies d_1' = d_2'$

$MS(\mathtt{AD},\mathtt{DA}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2) \implies d_1' = d_2'$

$MS(\mathtt{DA},\mathtt{AD}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2) \implies d_1' = d_2'$

$MS(\mathtt{DD},\mathtt{DD}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2) \implies d_1' = d_2'$

Relative termination conditions (A vs. B)

$RT(\mathtt{AD},\mathtt{DA}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2)$

$RT(\mathtt{DA},\mathtt{AD}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2)$

$RT(\mathtt{DD},\mathtt{DD}) : (x_1 = x_2 \wedge y_1 = y_2 \wedge d_1 = d_2)$

(c)

Fig. 14. Example of list traversal transformation. (a) Two different implementations `A`, and `B` of list traversal, (b) auxiliary procedures introduced during the proof, (c) MS and RT used to prove `A` and `B` equivalent.

procedures is an optimization for the purpose of equivalence checking; it avoids introducing the implication $((x_1 = x_2 \wedge g_1 = g_2) \implies r_1 = r_2 \wedge g_1' = g_2')$ explicitly in the formula to the theorem prover. However, the use of an uninterpreted function is restricted to modeling deterministic procedures, and only works when compared procedures have identical sets of arguments and globals.

List Traversal. Finally, we describe an example that requires careful interplay between mutual summary and relative termination specifications and well beyond the realm of present approaches using automated provers. Consider the two versions `A` and `B` of a program in Figure 14. Each version traverses elements in a list following the `next` field and updates the `data` field by an uninterpreted function U in the procedure `D`. The procedure `B` is a tail-recursive version of `A`. The transformation can be applied (either manually or by a compiler) to optimize the performance of the implementation. Preservation of semantics includes showing that the two versions diverge on the same inputs; it is easy to see that neither program terminates when the input is a cyclic list.

Although the change from A to B just swaps the order of calls to D and the recursive call, it has a global impact. Figure 15 demonstrates that the order of invoking the procedure D differs when A and B are invoked on the same input $u_0 \doteq \mathtt{x}, u_1 \doteq \mathtt{next}[u_0], \ldots, u_{k+1} \doteq \mathtt{next}[u_k]$. This makes proving the semantic equivalence of such transformations non-trivial. An intuition to understand the transformation from A to B is to think of creating intermediate programs that progressively transform an execution of A to an execution of B. Figure 15 shows the execution of such an intermediate program T that follows B's execution and then follows A's executions.

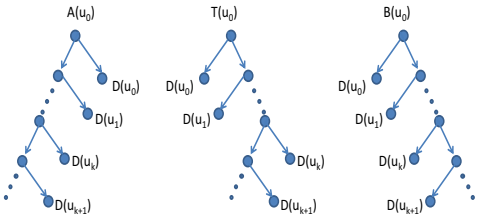


Fig. 15. Executions of different procedures over time

To handle such transformations, we need to provide a specification that allows *commuting* the calls to A and D. Such a specification can be obtained by introducing *composed procedures* AD, DA (Figure 14(b)) and writing mutual summary specifications on them. The procedure AD invokes A followed by D, and inlines the body (not nested callees) of A. The mutual summaries $MS(\mathtt{AD}, \mathtt{DA})$ and $MS(\mathtt{DA}, \mathtt{AD})$ (Figure 14(c)) express the fact that the summaries of AD and DA are equal on any input — in other words, A and D commute.

To leverage these auxiliary composed procedures in the proof, we have to relate the summary relations of these procedures (e.g. $R_{\mathtt{AD}}$) with that of the underlying procedures ($R_{\mathtt{A}}$ and $R_{\mathtt{D}}$). For a procedure fh composed of f and h, we automatically introduce the following axiom, which says that fh has a terminating execution if and only if f and h have a terminating execution through some intermediate global value g_2:

$$\forall x_1, x_2, g_1, g_1'.\ R_{fh}(x_1, x_2, g_1, g_1') \iff (\exists g_2.\ R_f(x_1, g_1, g_2)\ \wedge\ R_h(x_2, g_2, g_1'))$$

For this axiom to be sound, we require that at least one of f and h be inlined in fh. Although we do not yet have a formal proof of soundness for this axiom, we require the inlining for the axiom to fit within the inductive framework of Section 3.4. Intuitively, before the inductive step adds (x_1, x_2, g_1, g_1') to R_{fh}, the proof considers $R_{fh}(x_1, x_2, g_1, g_1')$ to be false, and thus requires that at least one of $R_f(x_1, g_1, g_2)$ and $R_h(x_2, g_2, g_1')$ be false for the axiom to hold, meaning that there cannot be calls in fh that introduce both the assumptions $R_f(x_1, g_1, g_2)$ and $R_h(x_2, g_2, g_1')$.

We need similar specifications for showing that D commutes with itself. Figure 14(c) contains all the mutual summary specification for this proof. The mutual summaries such as $MS(\mathtt{A}, \mathtt{A})$ are needed to express that A is *deterministic*,

a requirement to be able to prove the commute mutual summaries described above. Figure 14(c) also lists the relative termination conditions that were specified for this proof. Given the above mutual summaries and relative termination conditions (all of which express equality of inputs and outputs), we can show that all these specification are true to establish that if `A` and `B` start out with the same inputs and `A` terminates, then so does `B` with equal outputs.

5 Related Work

Our work is most closely related to work on proving equivalence in the context of compiler validation. Translation validation [9] is an approach for validating compilers by ensuring that each pair of source and target programs produced by the compiler are semantically equivalent. Necula [8] provided techniques to infer simulation relations by performing a lock-step analysis of the two programs, that generates simulation relations for simple compiler optimizations. Mutual summaries can capture such proofs that are based on establishing simulation relations. Zuck et al. [13] provide a rule PERMUTE that allows proving more complex optimizations that permute order of execution of loops (e.g. in loop reversal optimization). Tate et al. [11] provide an approach called *equality saturation* where an equality saturated program expression graph (PEG) can be used to capture equivalent programs. Tristan et al [12] instead provide rules for normalizing PEGs to perform translation validation. These approaches are automated and have been applied on various production compilers. Various domain specific languages (Cobalt [6], PEC [4]) have been devised to express compiler transformations as rewrite rules in a language. However, these approaches cannot validate interprocedural transformations (§ 4.3). Finally, the CompCert project [7] uses interactive theorem provers to provide an end-to-end correctness guarantee of semantic preservation by a compiler; this results in greater flexibility but less automation than approaches based on automated theorem provers. Pnueli and Zaks [10] generalize simulation-relation based translation validation to check simple interprocedural optimizations such as tail-recursion elimination, global constant propagation and inlining. However, program transformations such as translating a non tail-recursive procedure to its tail-recursive counterparts (Figure 14, Figure 12) will not be possible in this approach. Godlin and Strichman [3] describe automated methods for checking equivalence and mutual termination (under equal inputs) of mutually recursive procedures using uninterpreted functions as summaries. Our approach is not limited to proving equivalence but can be used to compare arbitrary mutual summaries. Mutual summaries provide more extensibility (at the cost of automation) by relating the summaries of two procedures with an arbitrary relation. This allows us to not only prove intraprocedural optimizations (that are not possible in [3]), but also new examples of interprocedural transformations (§4.3), including those that cannot be proved earlier (§6 in [3]). Relative termination allows reasoning about termination under specific conditions and generalizes the earlier work of checking mutual termination [3].

6 Conclusion

In this paper, we provided a general framework for comparing programs using program verifiers and automated theorem provers. We are currently working on extending the framework to handle more complex program transformations (e.g. the PERMUTE rule [13]), and automating the generation of mutual summary and the relative termination specifications. For most of the simple equality specifications used in this paper, we expect to leverage existing invariant synthesis techniques (e.g. predicate abstraction) to infer a majority of these specifications.

References

1. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Program Analysis For Software Tools and Engineering (PASTE 2005), pp. 82–87 (2005)
2. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
3. Godlin, B., Strichman, O.: Inference rules for proving the equivalence of recursive procedures. Acta Inf. 45(6), 403–439 (2008)
4. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Programming Language Design and Implementation (PLDI 2009), pp. 327–337. ACM (2009)
5. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: Symdiff: A language-agnostic semantic diff tool for imperative programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 712–717. Springer, Heidelberg (2012)
6. Lerner, S., Millstein, T.D., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: Programming Language Design and Implementation (PLDI 2003), pp. 220–231 (2003)
7. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Principles of Programming Languages (POPL 2006), pp. 42–54 (2006)
8. Necula, G.C.: Translation validation for an optimizing compiler. In: Programming Language Design and Implementation (PLDI 2000), pp. 83–94 (2000)
9. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
10. Pnueli, A., Zaks, A.: Validation of interprocedural optimizations. In: Compiler Optimization Meets Compiler Verification (COCV 2008) (2008)
11. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: Principles of Programming Languages (POPL 2009), pp. 264–276 (2009)
12. Tristan, J., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for llvm. In: Programming Language Design and Implementation (PLDI 2011), pp. 295–305 (2011)
13. Zuck, L.D., Pnueli, A., Goldberg, B., Barrett, C.W., Fang, Y., Hu, Y.: Translation and run-time validation of loop transformations. Formal Methods in System Design 27(3), 335–360 (2005)

Reuse in Software Verification by Abstract Method Calls*

Reiner Hähnle[1], Ina Schaefer[2], and Richard Bubel[1]

[1] Department of Computer Science, Technische Universität Darmstadt
{haehnle,bubel}@cs.tu-darmstadt.de
[2] Institute for Software Engineering, Technische Universität Braunschweig
i.schaefer@tu-braunschweig.de

Abstract. A major obstacle facing adoption of formal software verification is the difficulty to track changes in the target code and to accomodate them in specifications and in verification arguments. We introduce *abstract method calls*, a new verification rule for method calls that can be used in most contract-based verification settings. By combining abstract method calls, structured reuse in specification contracts, and caching of verification conditions, it is possible to detect reusability of contracts automatically via first-order reasoning. This is the basis for a verification framework that is able to deal with code undergoing frequent changes.

1 Introduction

Why is formal verification of software so much slower in industrial uptake than hardware verification? After all, it is expensive and requires special expertise for the hardware side, too. And on many occasions, software is safety-critical or at least errors are very expensive to fix, so there should be, a business case for formal software verification. But there is one decisive difference between soft- and hardware: software is, to a much greater extent than hardware, a moving target. Most application software is constantly *changed* to accomodate bug fixes or feature requests and to realize ever shorter time-to-market cycles. As compilation and deployment is cheap, this is no problem. But our current approaches to formal specification and verification do not support fast-paced changes: specification and verification effort is largely wasted when changes occur, systematic reuse is not possible (see Sect. 6 for a discussion). To improve the situation, two things are required: (i) formal specifications and verification proofs must be equipped with a systematic reuse principle; (ii) the reuse principle employed in the targeted code must match the one in specifications and proofs, so it is possible to reflect code changes when reasoning formally about them.

The standard composition principle of modern programming languages are procedure or method calls. To render formal verification scalable with the size

* Partly funded by the EU project FP7-231620 HATS (http://www.hats-project.eu) and by the German Science Foundation (SCHA1635/2-1)

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 300–314, 2013.

of target programs, most approaches use a formalization of Meyer's *design-by-contract* principle [15], where the behavior of a method is captured in the form of a contract between caller and callee. Contract-based specification has been realized for industrial target languages such as JAVA by specification languages such as JML [13] or in specific program logics, e.g., [2, Ch. 3]. The central idea of contract-based formal verification is to substitute a method call in the target code with a declarative specification of the effect of the call, obtained from the obligation that the callee ensures in its contract. For this to work, two things are necessary: first, the called method must have been successfully verified against its contract; second, the application requirements of the contract must be fulfilled in the call context. The problem of keeping up with target code changes during verification can be formulated in this framework as follows: Assume we have successfully verified a given piece of code p. Now, one of the methods m called in p is changed, i.e., m's contract in general is no longer valid. Therefore, this contract cannot be used in our proof of p which is accordingly broken and must be redone with the new contract of m. If p contains loops, then new invariants must be found, which is time consuming and expensive.

A rather restrictive approach to the change problem is *Liskov's principle* [14]: here, the new contracts must be substitutable for the old ones or, equivalently, only code changes that respect the existing contracts are permitted. But, even with optimizations [8,9], this is too restrictive in practice, because already very simple code modifications tend to break existing contracts. A more fundamental solution is called for, and this is the contribution of our paper.

Our approach consists of two elements: the first is a *structured reuse principle* for both code *and* contracts. Changes to target code and to contracts are expressed as "deltas" that describe explicitly the difference to the most recent version. Deltas in contracts foster reuse of specifications and make reused parts syntactically explicit. The details are in Sect. 3. But the problem remains that modified contracts are in general no longer applicable in a proof. It is impossible to figure out at method-call time whether a modified contract (or parts of it) might still be applicable or useful for the proof at hand. Therefore, the second ingredient of our approach is to *disentangle* in proofs the analysis of program code and the application of method contracts. This is achieved with *abstract contracts* in Sect. 4. In Sect. 5 we show that by combining abstract contracts, structured reuse of contract-based specification, and caching of first-order goals, it is possible to establish reusability of contracts by first-order reasoning. The result is a verification framework for programs under change, where reusable verification tasks are detected automatically.

2 Verification Framework

As mentioned above, we work in a contract-based [15] verification setting. Our approach is largely agnostic of the target language. Let us assume an imperative, class-based language with calls of methods whose implementation is known. (Resolving, for example, dynamic dispatch is an orthogonal problem to our concerns

and not considered here.) In the examples, we use a subset of JAVA. Our terminology follows closely that of KeY and JML [2,13]. We use an obvious notation to access classes `C` and methods `m` within a program $\mathcal{P}$: $\mathcal{P}$.`C`, $\mathcal{P}$.`C.m`, etc.

Definition 1. *A* program location *is an expression referring to an updatable heap location (variable, formal parameter, field access, array access). A* contract *for a method* `m` *consists of:*

1. *a first-order formula* `r` *called* precondition *or* requires clause*;*
2. *a first-order formula* `e` *called* postcondition *or* ensures clause*;*
3. *a set of program locations* `a` *(called* assignable clause*) that occur in the body of* `m` *and whose value can potentially be changed during execution.*

We extend our notation for accessing class members to cover the constituents of contracts: `C.m.r` is the requires clause of method `m` in class `C`, etc.

Definition 2. *Let* $\mathtt{m}(\overline{\mathtt{p}})$ *be a call of method* `m` *with parameters* $\overline{\mathtt{p}}$*. A* total correctness expression *has the form* $\langle \mathtt{m}(\overline{\mathtt{p}})\rangle\Phi$ *and means that whenever* `m` *is called then it terminates and in the final state* Φ *holds where* Φ *is either again a correctness expression or it is a first-order formula. (Partial correctness adds nothing to our discussion: we omit it for brevity.)*

In first-order dynamic logic [2] correctness expressions are just formulas with modalities. One may also encode correctness expressions as weakest precondition predicates and use first-order logic as a meta language, as typically done in verification condition generators (VCGs). Either way, we assume that we can build first-order formulas over correctness expressions, so we can state the intended semantics of contracts: Validity of the formula $\mathtt{r} \rightarrow \langle \mathtt{m}(\overline{\mathtt{p}})\rangle \mathtt{e}$ expresses the correctness of `m` with respect to the pre- and postcondition of its contract. In addition we must state correctness of `m` with respect to its assignable clause: one can assume [10] there is a formula $A(\mathtt{a}, \mathtt{m})$ whose validity implies that `m` can change at most the value of program locations in `a`. To summarize:

Definition 3. *A method* `m` *of class* `C` satisfies *its contract if the following holds:*

$$\models \mathtt{C.m.r} \rightarrow \langle \mathtt{m}(\overline{\mathtt{p}})\rangle \mathtt{C.m.e} \quad \wedge \quad A(\mathtt{C.m.a}, \mathtt{C.m}) \tag{1}$$

The presence of contracts makes formal verification of complex programs possible, because each method can be verified separately against its contract and called methods can be approximated by their contracts (see method contract rule below). The assignable clause of a method limits the program locations a method call can have side effects on.[1] To keep the treatment simple (and also in line with most implementations of verification systems), we do not admit metavariables to occur in first-order formulas.

[1] We are aware that this basic technique is insufficient to achieve modular verification. Advanced techniques for modular verification, e.g. [1], would obfuscate the fundamental questions considered in this paper and can be superimposed.

```
interface IAccount {
  Unit deposit(Int x);
  Unit withdraw(Int x);
  Unit move(Int x, IAccount a, IAccount b);
}
class Account implements IAccount {
  Int balance = 0;

  @requires x > 0;
  @ensures balance == \old(balance) + x;
  @assignable balance;
  Unit deposit(Int x) { balance = balance + x; }

  @requires a != b ∧ amount > 0;
  @ensures a.balance+b.balance <= \old(a.balance)+\old(b.balance)
  Unit move(Int amount, IAccount a, IAccount b) {
      a.withdraw(amount); // dual of deposit
      b.deposit(amount); } }
```

Fig. 1. Specification of a Bank Account Class

Example 1. Fig. 1 shows a simple bank account interface and its implementation. Contract elements appear before the method they refer to and start with a `@`. The method `deposit(x)` is specified with a contract whose precondition in the `@requires` clause says that the balance should be positive. The postcondition in the `@ensures` clause expresses that the balance after the method call is equal to the balance before the method call plus the value of parameter `x`. For simplicity, we use the JML keyword `\old` to access prestate values, but saving old values in renamed locations is equally possible. The obvious dual method `withdraw(x)` is not shown. Method `move(amount,a,b)` moves an amount between accounts. This should not increase the overall balance as stated in its `@ensures` clause.

In a calculus for verification of a method like `move(amount,a,b)` against its contract, methods calls (here, `deposit(x)` and `withdraw(x)`) must be replaced by their contracts using the *method contract rule* to achieve scalability:

$$\mathsf{methodContract}\ \frac{\Gamma \vdash \mathtt{m.r} \qquad \Gamma \vdash \mathcal{U}_{\mathtt{m.a}}(\mathtt{m.e} \rightarrow \langle\omega\rangle\Phi)}{\Gamma \vdash \langle \mathtt{m}(\overline{\mathtt{p}});\omega\rangle\Phi} \tag{2}$$

The rule is applied to the conclusion below the line: in a proof context Γ (a set of formulas) we need to establish correctness of a program starting with a method call $\mathtt{m}(\overline{\mathtt{p}})$ with respect to a postcondition Φ (typically, an ensures clause). We assume that the underlying verification calculus has associated the formal parameters of `m` with the actual parameters $\overline{\mathtt{p}}$. The rule uses the contract of `m` and reduces the problem to two subgoals. The first premise establishes that the requires clause is fulfilled, i.e., the contract is honoured by the callee. That

is exploited in the second premise, where the ensures clause can now be used to prove the remaining program ω correct. But one must be careful with the possible side effects the call might have had on the values of locations listed in the assignable clause of m's contract. As we cannot know these, we use a substitution $\mathcal{U}_{\mathtt{m.a}}$ to set all locations occurring in m.a to fresh Skolem symbols not occurring elsewhere (see [2, Sect. 3.8] for details). It is intentional that we do not commit to a specific calculus or program logic. Soundness of the method contract rules is formally stated here:

Theorem 1. *If the implementation of m satisfies its contract, then rule* (2) *is sound.*

Proof. The method contract rule is fairly standard except for the use of the substitution $U_{\mathtt{m.a}}$ which encodes the assignable clause of the contract. In [4] a theorem is shown from which the correctness of (2) follows as a special case.

Remark 1. In general, a JML contract may involve several *specification cases*, connected by the keyword **also**. It is possible to reduce this by propositional reasoning to proof obligations of the form that occur in the premisses of rule (2). So we assume wlog to deal with a single specification case at verification time. Similarly, each JML specification case may have multiple requires and ensures clauses, implicitly connected by conjunction. For simplicity, we assume wlog the presence of at most one requires/ensures clause per specification case.

The question we focus on in the following is: What happens with a correctness proof when the implementation of a called method changes? Liskov's well-known substitutability principle [14], rephrased in terms of contracts, gives one answer.

Definition 4 (*Substitutable Contract*). *For two methods* $\mathtt{m}$, $\mathtt{m}'$, *with contracts* $\mathtt{m.r}$, $\mathtt{m}'.\mathtt{r}'$, $\mathtt{m.e}$, $\mathtt{m}'.\mathtt{e}'$, $\mathtt{m.a}$, $\mathtt{m}'.\mathtt{a}'$, *the second method's contract is* substitutable *for the first if the following holds:*

$$(\mathtt{m.r} \rightarrow \mathtt{m}'.\mathtt{r}') \wedge (\mathtt{m}'.\mathtt{e}' \rightarrow \mathtt{m.e}) \wedge (\mathtt{m}'.\mathtt{a}' \subseteq \mathtt{m.a}) \tag{3}$$

The next lemma is immediate by the definition of contract satisfaction (Def. 3), propositional reasoning over (3), and monotonicity of postconditions in total correctness formulas.

Lemma 1. *If a method* $\mathtt{m}'$ *satisfies its contract, then it satisfies as well any contract substitutable for it.*

This justifies Liskov's principle and guarantees that a proof stays valid, whenever we replace a method by one whose contract is substitutable for it. As we shall soon see, substitutability is much too restrictive to be practically useful.

3 Delta-Oriented Reuse of Programs and Contracts

If a program evolves due to bug fixes, newly added features or other modifications, the program code itself and also its specification in form of method

```
delta DFee(Int fee);
modifies class Account {
  modifies Unit deposit(Int x) { if (x>=fee) original(x-fee); } }
```

Fig. 2. Delta for introducing a fee to the bank account

contracts changes. To enable systematic reuse for verification, we need to represent changes explicitly. To this end we use the ideas from *delta-oriented programming* [17,6] (DOP), a code reuse technique originally developed for implementing static variability in software product line engineering. A *delta* is simply a container of descriptions of modifications applied to a program or a specification. It can be thought of as the "diff" between subsequent versions of a program and its associated contracts. Hence, deltas are a flexible concept that can be used to represent anticipated and unanticipated changes of programs and specifications *in a structured manner*. The case for DOP is made elsewhere [6,17]. Let it suffice to say that deltas achieve a separation of concerns between the modelling of variability and the modelling of data and code design, which are conflated in standard OO languages.

3.1 Program Deltas

Different flavours of DOP can be found in the literature, but they have in common that deltas may add, remove, and modify elements of the target program. At the class level, we use the keywords **modifies**, **adds**, and **removes** preceding the changed class declaration. With **adds**, new classes can be created and with **removes** a whole class can be removed. A modified class contains further directives that may change its method and field declarations, for which the same keywords are used (for fields, only **adds** and **removes** are permitted). In general, the **modifies** directive is a mere convenience, as it can be replaced by a suitable combination of **removes** and **adds**.

A modified method declaration can be completely replaced or it can be a *wrapper* using the **original** call. The keyword **original** stands for a call to the most recent version of the currently modified method and it must match its type signature. The **original** construct makes code reuse possible and is reminiscent of **super** calls in standard OO languages. A main difference between **original** and **super** is that the former, as all other changes contained in a delta, are resolved at compile time, when applying a delta to an existing program. The same method can be modified and wrapped in several subsequent delta applications capturing individual changes.

Example 2. We want to extend the account class of Ex. 1 with a feature to charge a fee for each deposit. This is realized in the delta in Fig. 2. Delta declarations begin with the keyword **delta**, followed by a name and optional parameters, here, the amount of the transaction fee. The delta modifies class `Account` and its method `deposit(x)` by wrapping and reusing the previous version with an

```
product AccountWithFee(Base, DFee(2));
// results in:
interface IAccount { Unit deposit(Int x); }
class Account implements IAccount {
  Int balance = 0 ;
  Unit deposit(Int x) { if (x>=2) balance = balance + (x-2); } }
```

Fig. 3. Result of applying the *DFee* delta to the bank account (shown partially)

original call. The conditional around the call to **original** ensures that the deposited amount is not lower than the fee to avoid counter-intuitive results.

Obviously, the application of a delta to a given program may fail. The **modifies** and **removes** directives implicitly assume the existence of program elements that may be missing, the type signature of an **original** call may not match, etc. Therefore, compilation in DOP is a two-stage process: in a first step, deltas are applied in a user-specified sequence, where the well-definedness of each delta application is checked, and the result is a flattened, delta-free program to be processed further with a standard compiler. To specify products resulting from delta applications, we use the keyword **product**, followed by a name and a sequence of the deltas that are to be applied.

Example 3. Fig. 3 shows the declaration of a bank account product with deposit fee derived from Ex. 1 (for which the name `Base` is used by convention) and subsequent application of the `DFee` delta of Ex. 2, where the parameter is instantiated with 2 units. The declaration of class `Account` is generated automatically by the delta compiler.

3.2 Contract Deltas

Changing program code typically requires changing method contracts since a change might intentionally cause a different functionality that has to be reflected in the contract. Hence, it is natural to extend the concept of deltas to contracts. Following [5,12], we permit the keywords **adds**, **modifies**, and **removes** in front of specification cases and requires/ensures/assignable clauses. If there is more than one specification case we use names to distinguish them. A name clause is of the form "`@case <name>;`" and this name can be used to qualify a **modifies** or **removes** directive.

Going beyond [5], and in analogy to program deltas, we allow reuse of specifications in modified and added contract clauses using the keyword **original**. As with program deltas, this means that the most recent version of the contract clause is replaced by the **original** keyword when the delta is applied. If there is more than one specification case, **original** can be qualified with a name.

Example 4. For the contract of the modified `deposit(x)` method in `DFee` (Fig. 2), we want to reuse the contract of the **original** method shown in Fig. 1. This can

```
delta DFee(Int fee);
modifies class Account {
  // modify the only existing specification case
  modifies @requires \original ∧ x >= fee;
  modifies @ensures balance == \old(balance) + x - fee;
  // add a new specification case
  adds also @case TrivialAmount
            @requires \original ∧ x < fee;
            @assignable \nothing;
  modifies Unit deposit(Int x) { if (x>=fee) original(x-fee); }

  modifies @requires \original ∧ amount>=fee
  of Unit move(Int amount, IAccount a, IAccount b) }
```

Fig. 4. Delta changing the specification of the deposit method

```
@requires x > 0 ∧ x >= 2;
@ensures balance == \old(balance) + x - 2;
@assignable balance;
also
@case TrivialAmount
@requires x > 0 ∧ x < 2;
@assignable \nothing;
Unit deposit(Int x) { if (x>=2) balance = balance + (x-2); }
```

Fig. 5. Result of applying the DFee program and contract delta

look as in Fig. 4. Note that we need two specification cases: one when the fee does not exceed the deposited amount and one when it does. The first is obtained as a modification of the existing contract: in the requires clause, a suitable precondition is added to the **original** requires clause. The previous version of the ensures clause is replaced by a new version which takes the deduction of the fee into account. The assignable clause is untouched. The second case is obtained by **adds**. Again, the **original** precondition is reused. In this case, the balance of the account remains unchanged, which is implied by the new assignable clause. While this is sufficient, it is hard to detect and, therefore, to exploit.

Example 5. If we compute the product shown in Fig. 3 by delta application *including* the base contract and the contract of the delta we obtain the contract of the modified `deposit(x)` method shown in Fig. 5.

3.3 Verification of Deltas

The main question in a formal verification context is: *how to prove that a program delta satisfies its contract delta?* In general, this is not possible for a delta in isolation, for two reasons: the first is, before actually applying a delta, its code

and its contract are partially unknown. In a previous paper [12] we addressed this issue by imposing a number of constraints: occurrences of **original** in requires clauses must be of the form **original** $\vee$ `r'`, in ensures clauses of the form **original** $\wedge$ `e'`, and in assignable clauses of the form **original**`\a'`. This ensures substitutability of reused contracts and makes Lemma 1 applicable. The second problem is that a method satisfying its contract might cease to do so after modification of either code or contract. In [12] we imposed two conditions relative to a given partial order $\prec$ of delta applications: (i) modified contracts in $\prec$-larger deltas must be substitutable, and (ii) calls to methods that might have been modified are replaced with the $\prec$-minimal contract of that method. Under these conditions, satisfaction of each contract in the base and in all deltas implies that all contracts in any $\prec$-compatible product are satisfied [12, Thm. 1]. Unfortunately, these restrictions are too severe in practice:

Example 6. Consider the contracts in Figs. 1, 4. The ensures clause of the modified contract introduces the parameter `fee` and bears no logical relation to the clause it replaces. In the added specification case, the implicitly given default ensures clause "`@ensures true;`" is *weaker*, not stronger as required.

4 Abstract Method Calls

Ex. 6 shows that already minor changes to programs violate Liskov's principle. This makes reuse of verification effort problematic in general:

Example 7. Consider method `move()` from Fig. 1 which transfers money between two accounts. Its contract states that money might be lost, because of fees or similar, but it strictly excludes generation of money. Its implementation calls `deposit()` to credit the receiving account. To prove that `move()` satisfies its contract requires applying the method contract rule (2) to `deposit()`. Changing the contract of `deposit()` to the version in Fig. 5, entails that neither of the two specification cases satisfies Liskov's principle (see Ex. 6). Consequently, in the verification of `move()` no reuse is possible.

The previous example highlights an unfortunate property of the usual verification setup: assuming we use a complete[2] verification method, intuitively, the logical information in the proof with the call to the original `deposit()` should be sufficient to justify the contract of `move()` even for the version with a fee. But this cannot be detected easily, because the proof of `move()`'s contract uses the ensures clause of `deposit()`. Now, when `deposit()`'s contract is changed, it is impossible to disentangle the information from `deposit()`'s contract and the steps used to prove `move()`. To achieve such a separation we need a technique that splits method invocation from the actual contract application. This is the central contribution of this paper and explained now.

The main technical idea is to introduce a level of indirection into a method contract that allows us to delay the substitution of its concrete requires and

[2] Relatively complete, with respect to Peano arithmetic, of course.

```
// abstract contract
@requires R;
@ensures l1 == \def(l1) && ... && ln == \def(ln) && E;
@assignable l1, ..., ln;
// definitions allowing us to '' instantiate '' the contract
@def R = R', \def(l1) = E1, ..., \def(ln) = En, E = E';
```

Fig. 6. Shape of an *abstract method contract*

```
@requires R;
@ensures balance == \def(balance);
@assignable balance;
@def R = x > 0, \def(balance) = \old(getBalance()) + x;
Unit deposit(Int x) {...}
```

Fig. 7. Abstract method specification for `deposit` without fee

ensures clause. We call this an *abstract method contract*. It has the shape shown in Fig. 6. Its *abstract* section consists of the standard requires, ensures, and assignable clauses. As before, the assignable clause specifies all locations that might be changed by the specified method. The requires and ensures clauses, however, now merely contain placeholders `R`, `E`, and $\texttt{\def(l}_\texttt{i}\texttt{)}$, which are defined by concrete formulas and terms in the *definitions* section, which must contain a definition for each placeholder in the abstract section. The ensures clause is a conjunction of equations specifying the post value of each possibly changed location in the assignable clause and additional properties `E` on the post state. Please note that Fig. 6 is merely a convenient notation. The formal definition of an abstract method contract is given in Def. 5 below.

The main restriction inherent to abstract method contracts is that the assignable clause explicitly lists updatable locations (i.e., it is not abstract). Nevertheless, it is part of the abstract section, so that it is shared by all clients of the contract. This is necessary to ensure that applying any abstract contract for a method has the same result before definitions are unfolded. Another, minor restriction is the equational form, which enforces that the post value for any assignable location is well-defined after contract application. Field accesses occurring in definitions are expressed using getter methods, e.g., `getBalance()` is used to access the `balance` field. This ensures that their correct value is used when definitions are unfolded.

Example 8. Fig. 7 reformulates the contract of method `deposit()` in terms of an abstract method contract.

Abstract method contracts are *fully compatible* with contract deltas, with the restriction that assignable clauses may not be changed. The only difference is that all changes specified in a delta are acted upon in the *definition* section of an abstract contract—the abstract section remains completely unchanged. In our

```
@assignable balance;
@requires R;
@ensures balance == \def(balance);
@def R = \original_Base(R) && x>=2, \def(balance) = \old(getBalance())+x-2;
Unit deposit(Int x) {...}
```

Fig. 8. Abstract method specification case for a successful `deposit()` with fee

example, the application of the delta in Fig. 4 results in an abstract contract with two specification cases, one of which is shown in Fig. 8.[3]

It is perhaps surprising that **original** still occurs after delta application. The explanation is that the abstract shape of contracts does not force us anymore to unfold **original** references immediately. As we shall see in Sect. 5, it can have advantages not to do so. To indicate that the **original** has been in fact resolved, we add a reference (here: `Base`) to its container. Now we can define abstract method contracts formally:

Definition 5 (Abstract method contract). *An* abstract method contract C_m *for a method* m *is a quadruple* $(r, e, U, defs)$ *where*

- r, e *are logic formulas representing the contract's pre- and postcondition,*
- U *is an explicit substitution representing the assignable clause, and*
- $defs$ *is a list of pairs* $(defSym, \xi_{defSym})$ *where* $defSym$ *are non-rigid (i.e., state dependent) function or predicate symbols used as placeholders in* r, e, *and* ξ *their defining term or formula. For each* $\texttt{\textbackslash def}(\texttt{l}_\texttt{i})$ *there is a unique function symbol in* $defs$*. For simplicity, we refer to both with* $\texttt{\textbackslash def}(\texttt{l}_\texttt{i})$*, as long as no ambiguity arises.*

Placeholders must be non-rigid to prevent the program logic calculus to perform simplifications over them that are invalid in some program states. To ensure soundness of the abstract setup we add the definitions of the placeholders (i.e., the contents of the definition section of each abstract contract) as a *theory* to the logic, just like other theories, such as arithmetic, etc. This means that the notion of contract satisfaction (Def. 3) is now able to consider defined symbols in abstract contracts. Additionally the rule on the right that substitutes placeholders by their definitions (by a slight abuse of notation, but with obvious meaning for function symbols) is now obviously sound. The advantage of this setup is that we can still use the old method contract rule (2), which simply ignores the definition section. As we changed neither the satisfaction of contracts nor the method contract rule, Thm. 1 still holds.

expandDef

$$\frac{\xi_{defSym}}{defSym}$$

[3] There is a technicality here about represening multiple specification cases for abstract method contracts. This is inessential and distracting, so we don't give details.

5 Application Scenarios

5.1 Abstract Verification

Rule (2) now uses only to abstract section of an abstract method contract, but is otherwise completely unchanged. Hence, its application yields the same result for all method contracts that share the same abstract section. This allows us to define the following general verification process: Assume we want to establish that a method m satisfies a contract C. In the first phase the rules of the underlying calculus are applied until only first-order proof goals are remaining. This can be done, for example, with symbolic execution or with a VCG and typically involves manual annotation of the target program with suitable loop invariants. During this phase all calculus rules, including SMT and first-order solvers, except for expandDef may be used to close subgoals. If the implementation of m contains a call to another method n we use an abstract contract C_n for the latter. Because of this, some first-order subgoals will usually remain open. Let us call this partial proof p.

There are two things one can do at this stage: first, we can use the expandDef rules of the definition section of C_n and first-order reasoning on the open subgoals of p. If m satisfies C and suitable invariants were chosen in p, this completes the proof by first-order reasoning. Second, assume now we made changes to n and modified the definition section of its contract, let's call it C'_n. As long as C_n and C'_n have the same abstract section, we can *reuse* proof p completely. To test whether p still satisfies C after the change, it is sufficient to use the expandDef rules of the definition section of C'_n on p. Again, this is a first-order problem. This is significant, because coming up with the right invariants is usually much more expensive than first-order reasoning.

Example 9. Applying the verification rules of KeY [2] to show that `move()` satisfies its contract (Fig. 1), while using abstract contracts of `deposit()` and `withdraw()`, results in a partial proof p with the open first-order goal

$$\texttt{\textbackslash def(a.balance)} + \texttt{\textbackslash def(b.balance)} \leq \texttt{\textbackslash old(a.balance)} + \texttt{\textbackslash old(b.balance)}$$

If we use the expandDef rules gained from the definition sections (cf. Fig. 7) we obtain the goal

$$\texttt{\textbackslash old(a.getBalance())} - \texttt{x} + \texttt{\textbackslash old(b.getBalance())} + \texttt{x} \leq \texttt{\textbackslash old(a.balance)} + \texttt{\textbackslash old(b.balance)}$$

which is trivial for an SMT solver. In addition, we can *reuse* p after applying the `DFee` delta to `deposit()`, because the abstract contracts in Figs. 7, 8 have identical abstract sections. With the rules gained from the definition section of Fig. 8 the resulting subgoal is still first-order provable.

5.2 Liskov for Free

A nice feature of our approach is that preservation of changed contracts as justified by Liskov's substitutability principle (Sect. 3.3) is detected automatically.

Let n' be a method whose contract $C_{n'}$ is substitutable (3) for n's contract C_n and assume m invokes n. Abstract verification first constructs a partial proof p_m for m and its contract that has open first-order verification conditions V_m. These contain placeholders from C_n. Assume we can finish p_m by expanding their definitions plus first-order reasoning.

To verify that m still satisfies its contract when n is replaced with n', we proceed as follows: in V_m substitute each placeholder from C_n with its corresponding placeholder from $C_{n'}$. This is possible, because both contracts have identical abstract sections. After expanding the definitions, by substitutability (3), one first-order subsumption step is enough to obtain the definitions from C_n, which have been proven already. Therefore, (complete) first-order reasoning will automatically detect such a situation.

5.3 Experiments

We performed preliminary experiments using the KeY verification system[4]. The necessary abstract method contracts and definition expansion rules have been provided manually. Manual steps for saving and loading of the partial proof were also necessary, but will be fully automatic once proper support for abstract method calls is implemented.

The example used in this paper showed only modest gains by abstract method calls, which is not surprising considering the low complexity of the involved methods. Verifying the more complex method `requestTransaction()` which calls `deposit()` and contains a loop, we achieved savings of 90%. Savings refer to the ratio of $\frac{\textit{partial proof size}}{\textit{proof size}}$ (proof size measured in number of nodes (branches)):

Example	Partial Proof	Proof (`Base`)	Savings	Proof (`DFee`)	Savings
move	100 (6)	477 (10)	21%	517 (10)	19%
reqTrans	887 (20)	976 (23)	91%	979 (23)	91%

5.4 Program Evolution

For verifying a program that evolves by changing methods via delta operations, we can proceed as follows: For each contract C_m of each method m contained in the initial program, we construct a proof (e.g., by VCG or symbolic execution) using the abstract method contracts. We store the proof and also the open subgoals in a cache for future reference. Then, we unfold the definitions in the abstract method contract C_m and use, e.g., an SMT solver to close the open subgoals to verify the method.

If the program evolves by delta application, we consider several cases: If the implementation of a method m and its contract C_m as well as all abstract sections of contracts C_n for methods n called by m remain unchanged, we can completely reuse the stored partial proof for C_m as in Sect. 5.1. For contracts of called methods n that are unchanged, we can reuse previous proof goals stored in the

[4] The experiments are available at `http://www.key-project.org/cade13/tud/`

cache. If the contracts of the called methods n are changed, but substitutable for C_n, we obtain the proof as described in Sect. 5.2. If the contracts of called methods are changed in other ways, we unfold their definitions and obtain the proof by first-order reasoning and store the new proof goals in the cache. If the implementation of method m or its contract changed, we need to construct a new proof for C'_m as for the initial program. Here, we do not unfold the definitions of **original** when the partial proof is stored in the cache in order to be able to reuse partial proofs also for different instantiations of original. If, in the newly constructed proof, contracts for methods n called by m did not change or their contracts are substitutable wrt. previous contracts, we can reuse proof goals stored in the cache or apply the principle of Sect. 5.2. If a method is newly added, it has to be verified from scratch and the proof is stored in the cache.

6 Related Work

Previous work on deductive verification of evolving programs [3,16] proposes proof replay to ameliorate verification effort. The old proof is replayed; when this is no longer possible, a new proof rule is guessed. Paper [3] uses a similarity function to control replay, while [16] uses differencing operations. Unlike our work, proof replay is unrelated to the program or specification reuse principle.

In [9], a set of allowed changes to evolve an OO program is introduced which is similar to delta operations. For verified method contracts, a proof context is constructed which keeps track of proof obligations. Program changes cause the proof context to be adapted so that the proof obligations that are still valid are preserved and new proof goals are created. The proof context is similar to the proof cache proposed in Sect. 5.4, but reuse only happens at the level of contracts, not on the level of (partial) proofs as in our work. Earlier work along the same lines in the context of VCG is [11].

There is some recent work targeting efficient verification of delta-oriented programs in the context of software product line engineering, where *static* program variability is considered, in contrast to program evolution, which is considered here. In [5], it is assumed that one program variant has been fully verified. From the structure of a delta to generate another program variant it is analyzed which proof obligations remain valid in the new product variant and need not be reestablished. The main result of [12], and its restrictions, are discussed in Sect. 3.3. In [7], methods in a delta are verified based on a contract which makes assumptions on the contracts of the called methods explicit. The main difference to our work is that reuse in the approaches above only happens at the level of the proof obligations limiting their reuse potential.

7 Conclusion

We presented a framework for systematic reuse of verification effort for programs and specification under change. Its distinctive feature is that reuse takes place at the level of code, specification, and proofs with a matching reuse principle. Our

work highlights the importance of first-order ATP in verification of programs that undergo frequent changes. Detaching the *usage* from the *validation* of contracts turns the test for reusability of previously cached results from a specific verification problem into a general first-order problem. A logical next step is to investigate the nature and the complexity of the resulting first-order problems.

References

1. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6), 27–56 (2004)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
3. Beckert, B., Klebanov, V.: Proof reuse for deductive program verification. In: SEFM, pp. 77–86. IEEE Computer Society (2004)
4. Beckert, B., Schmitt, P.H.: Program verification using change information. In: Proceedings, Software Engineering and Formal Methods (SEFM), Brisbane, Australia, pp. 91–99. IEEE Press (2003)
5. Bruns, D., Klebanov, V., Schaefer, I.: Verification of software product lines with delta-oriented slicing. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 61–75. Springer, Heidelberg (2011)
6. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 417–457. Springer, Heidelberg (2011)
7. Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E.B., Yu, I.C.: A transformational proof system for delta-oriented programming. In: SPLC (2), pp. 53–60 (2012)
8. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Lazy behavioral subtyping. Journal of Logic and Algebraic Programming 79(7), 578–607 (2010)
9. Dovland, J., Johnsen, E.B., Yu, I.C.: Tracking behavioral constraints during object-oriented software evolution. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 253–268. Springer, Heidelberg (2012)
10. Engel, C., Roth, A., Schmitt, P.H., Weiß, B.: Verification of modifies clauses in dynamic logic with non-rigid functions. Technical Report 2009-9, University of Karlsruhe (2009)
11. Grigore, R., Moskal, M.: Edit & verify. In: First-order Theorem Proving Workshop, Liverpool, UK (2007), http://arxiv.org/abs/0708.0713v1
12. Hähnle, R., Schaefer, I.: A Liskov principle for delta-oriented programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 32–46. Springer, Heidelberg (2012)
13. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M.: JML Reference Manual, Draft (September 2009)
14. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16(6), 1811–1841 (1994)
15. Meyer, B.: Applying "Design by Contract". IEEE Computer 25(10) (October 1992)
16. Reif, W., Stenzel, K.: Reuse of proofs in software verification. In: FSTTCS, pp. 284–293 (1993)
17. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)

Dynamic Logic with Trace Semantics

Bernhard Beckert and Daniel Bruns*

Karlsruhe Institute of Technology (KIT), Germany

Abstract. Dynamic logic is an established instrument for program verification and for reasoning about the semantics of programs and programming languages. In this paper, we define an extension of dynamic logic, called Dynamic Trace Logic (DTL), which combines the expressiveness of program logics such as dynamic logic with that of temporal logic. And we present a sound and relatively complete sequent calculus for proving validity of DTL formulae.

Due to its expressiveness, DTL can serve as a basis for proving functional and information-flow properties in concurrent programs, among other applications.

1 Introduction

Overview. Dynamic logics (DL) [8] are multi-modal first-order logics where each legal sequential program fragment π (i.e., a sequence of statements) gives rise to a modal operator $[\pi]$. The formula $[\pi]\varphi$ expresses "in any state in which π terminates, φ holds." An interesting special case are deterministic programing languages, for which there is at most one terminal state. Program logics like DL are more expressive than Hoare logics in that programs are part of formulae, which can be self-composed. This allows, for instance, to express information-flow properties such as non-interference [12]. In other regards, however, standard dynamic logic lacks expressiveness: The semantics of a program is a relation between states; formulae can only describe the input/output behaviour of programs. It is inadequate for reasoning about non-terminating programs and for verifying temporal properties.

To combine the advantages of dynamic logic and temporal logic, our Dynamic Trace Logic uses trace-based program semantics and the well-known temporal operators $\Box$ (always), $\Diamond$ (eventually), $\bullet$ (weak next), $\circ$ (strong next), $\mathbf{U}$ (until), $\mathbf{W}$ (weak until), and $\mathbf{R}$ (release) similar to those of Linear Temporal Logic (LTL). In DTL, the formula $[\![\pi]\!]\varphi$ expresses that φ holds for the (possibly infinite) trace of the program π when started in the current state. For example, the formula

$$[\![\pi]\!]\Box\forall u.\forall v.(X \doteq u \wedge \circ(X \doteq v) \rightarrow u \leq v)$$

is a two-state invariant. It says that the value of the program variable X must increase or remain the same throughout the trace of π. Proving such two-state

* This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under project "Program-level Specification and Deductive Verification of Security Properties (DeduSec)" within SPP 1496 "Reliably Secure Software Systems (RS^3)".

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 315–329, 2013.

invariants is the basis of the rely-guarantee approach for verifying concurrent programs.

Target Programing Language. In the following, we use a simple while language as target programing language without method calls or any feature of object-orientation. However, our language distinguishes between local variables with state-internal assignments and global variables with assignments inducing state transitions. The rationale behind this is that, in a concurrent setting, only global variables can be observed by the environment.

Of course, to be useful in practice, DTL needs to be extended to real-world programing languages. The KeY verification system (co-developed by the authors) is built on a calculus for JavaDL, a dynamic logic for sequential Java [3,5]. This has been used as a basis to extend DTL to Java and implement the DTL calculus (a prototypical implementation exists). Additional rules needed to handle full (sequential) Java can be derived from the KeY rules for the $[\cdot]$ modality by analogy. Since a language like Java incorporates a lot of features, in particular object-orientation, and various syntactic sugars, the rule set is rather voluminous in comparison to simple while languages. These special cases can, however, be reduced to a smaller set of base cases. For instance, the assignment `x=y++` containing a post-increment operator is transformed into two consecutive assignments `x=y` and `y=y+1` during symbolic execution.

Related Work. In earlier work [6], we have extended Dynamic Logic with a modality also written $[\![\cdot]\!]$, where $[\![\pi]\!]\varphi$ stands for "φ holds throughout the execution of π." This can be seen as a special case of DTL because the same property can be expressed in DTL as $[\![\pi]\!]\Box\varphi$. That is, in our earlier work, the temporal formula was restricted to the form $\Box\varphi$ with φ not containing further temporal operators. Platzer [10] introduced Temporal Dynamic Logic (dTL), where programs are *hybrid programs*; in particular, they are indeterministic, and therefore, traces are branching. It features formulae of the shapes $[\![\pi]\!]\Box\varphi$ ("for all traces, φ always holds") and $\langle\!\langle\pi\rangle\!\rangle\Diamond\varphi$ ("there is a trace such that eventually φ holds") where φ is a state formula. There is no further combination of temporal operators. Similar to our setting in this paper, traces can be of finite or infinite length. Platzer presents a sequent calculus for dTL, which, however, is incomplete, much due to the continuous state space of hybrid programs.

Reasoning about temporal properties is traditionally the domain of model checking. Nevertheless, there is some work on deductive techniques (tableaux, sequent calculi, resolution etc.) applied to temporal logics. Good sources on the topic of theorem proving for propositional linear-time logics are an article by Wolper [15] and the textbook chapters by Goré [7] and Reynolds and Dixon [11]. The work by Wolper introduces a tableau method for propositional LTL. A calculus for first-order LTL has been presented by Abadi and Manna [1]. It is known that, although LTL is decidable, there does not always exist a finite proof tree. The proof graph may contain cycles in the presence of eventualities (i.e., formulae with a positive occurrence of $\mathbf{U}$). There are different techniques to deal with this. In the calculus presented in this paper, we use program invariants.

Language-based program verification is usually done w.r.t. state or two-state formula (pre and post). Program verification w.r.t. temporal specifications has been considered by Schellhorn et al. [13], where programs themselves are formulae of Interval Temporal Logic (ITL) [9]. In an earlier work, they have presented a sequent calculus for ITL [14], which allows to prove the correctness of programs w.r.t. ITL specifications.

Structure of this Paper. Syntax and semantics of our logic DTL are defined in Sects. 2 resp. 3 (including syntax and semantics of the while language that we use as the target programing language in this paper). In Sect. 4, we present our sequent calculus for DTL. Notions of soundness and completeness are defined in Sect. 5, and we sketch soundness and completeness proofs. Complete proofs can be found in an extended version of this paper [4].

2 Syntax of DTL

Signatures and Expressions. We assume disjoint sets *LVar* of local program variables and *GVar* of global program variables to be given. In addition, there is a set V of logical variables. Logical variables are rigid, i.e., they cannot be changed by programs and – in contrast to program variables – are assigned the same value in all states of a program trace. Quantifiers can only range over logical variables and not over program variables. In this paper, the sets of function and predicate symbols are fixed. They only contain the usual integer and boolean operators with their standard semantics.

Definition 1 (Expressions). Expressions *of type integer are constructed as usual over integer literals, local and global variables, logical variables, and the operators* $+, -, *$. *Expressions of type boolean are constructed using the relations* $\doteq, >, <$ *on integer expressions, the boolean literals true and false, and the logical operators* $\wedge, \vee, \neg$.

Programs. Programs are written in a simple while language, with the (mathematical) integers as the only data type. Expressions can be of types integer and boolean; they do not have side-effects. The program language does not contain features such as functions and arrays; and there are no object-oriented features. As discussed above, all such features can be added, but we keep the programing language simple for the presentation in this paper.

The only special feature is the distinction between local variables (written in lowercase letters) and global variables (written in uppercase). As will be explained in Sect. 3, we consider assignments to global variables to be the only program statements that lead to a new observable state. As a technical restriction, to ensure that there cannot be a program that gets stuck in an infinite loop without ever progressing to a new observable state, we demand that in every loop execution, an assignment to a global variable is executed.[1]

[1] This property is undecidable in general, but a sufficient syntactical criterion could be that every possible execution path contains an assignment (which may be ineffective, e.g., `X = X;`).

Definition 2 (Statements, programs). Programs *and* statements *are inductively defined, where statements are of the form:*

- `x = a;` *where* `x` $\in$ *LVar and* `a` *is an expression of type integer* (assignment to local variable),
- `X = a;` *where* `X` $\in$ *GVar and* `a` *is an expression of type integer* (assignment to global variable),
- `if (a) {`π_1`} else {`π_2`}` *where* `a` *is an expression of type boolean not containing logical variables and* π_1 *and* π_2 *are programs* (conditional), *or*
- `while (a) {`π`}` *where* `a` *is an expression of type boolean not containing logical variables and* π *is a program that contains at least one assignment to a global variable on every execution path* (loop).

Programs are finite sequences of statements. The empty program is denoted by ϵ.

State Updates. An important property of the calculus for DTL presented in Sect. 4 (as well as the calculus for JavaDL used in the KeY System) is that programs are *symbolically executed* starting from an initial state – in contrast to *wp*-calculi where one starts with a postcondition and works in a backwards manner. In order to capture the state transitions in between, we use a prefix on formulae, called *state update*. Updates can be thought of as "delayed substitutions," i.e., a substitution takes place once the program has been completely eliminated.

Definition 3 (State updates). *Let* x *be a (local or global) program variable, and let* a *be an expression. Then,* $\{x := a\}$ *is an update.*

For instance, $\{x := 4\}$ and $\{x := x + 1\}$ are updates. Applying these updates (after each other, from right to left) to the formula $x \doteq 5$ yields $4 + 1 \doteq 5$.

DTL Formulae. Formulae have the general appearance $\mathcal{U}[\![\pi]\!]\varphi$ where $\mathcal{U}$ is a sequence of updates, π is a program, and φ is a formula (that may or may not contain temporal operators and further sub-formulae of the same form). Intuitively, $\mathcal{U}[\![\pi]\!]\varphi$ expresses that φ holds when evaluated over all traces τ such that the initial state of τ is (partially) described by $\mathcal{U}$ and the further states of τ are constructed by running the program π.

Definition 4 (Formulae). State formulae *and* trace formulae *are inductively defined as follows:*

0. *All boolean expressions (Def. 1) are state formulae.*
1. *All state formulae are also trace formulae.*
2. *If* φ *and* ψ *are (state or trace) formulae, then the following are trace formulae:* $\Box\varphi$ *(always),* $\bullet\varphi$ *(weak next),* $\varphi \,\mathbf{U}\, \psi$ *(until).*
3. *If* $\mathcal{U}$ *is an update and* φ *a state formula, then* $\mathcal{U}\varphi$ *is a state formula.*
4. *If* π *is a program and* φ *a trace formula, then* $[\![\pi]\!]\varphi$ *is a state formulae.*
5. *The sets of state and trace formulae are closed under the logical operators* $\neg, \wedge, \forall$.

In addition, we use the following abbreviations:

$$\begin{aligned} \Diamond\varphi &:= \neg\Box\neg\varphi, & \circ\varphi &:= \neg\bullet\neg\varphi, \\ \varphi\,\mathbf{W}\,\psi &:= \varphi\,\mathbf{U}\,\psi \vee \Box\varphi, & \varphi\,\mathbf{R}\,\psi &:= \neg(\neg\varphi\,\mathbf{U}\,\neg\psi), \\ \varphi \vee \psi &:= \neg(\neg\varphi \wedge \neg\psi), & \varphi \to \psi &:= \neg\varphi \vee \psi, \\ \exists x.\varphi &:= \neg\forall x.\neg\varphi. \end{aligned}$$

A formula is called non-temporal *if it neither contains a temporal operator nor a program modality* $[\![\pi]\!]$.

3 Semantics of DTL

Expressions and formulae are evaluated over traces of states (which give meaning to program variables) and variable assignments (which give meaning to logical variables). The domain of DTL is always $\mathbb{Z}$, irregardless of the state (*constant domain*).

Definition 5 (States, variable assignments). *A* state *s is a function assigning integer values to all local and global variables, i.e., $s : LVar \cup GVar \to \mathbb{Z}$.*

A variable assignment *β is a function assigning integer values to all logical variables, i.e., $\beta : V \to \mathbb{Z}$.*

We use the notation $s\{x \mapsto d\}$ to denote the state that is identical to s except that the variable x is assigned the value $d \in \mathbb{Z}$. Likewise, we write $\beta\{x \mapsto d\}$ and $\tau\{x \mapsto d\}$ (where τ is a trace, see below) with the obvious semantics.

Definition 6 (Traces). *A* trace *τ is a non-empty, finite or infinite sequence of (not necessarily different) states.*

We use the following notations related to traces: (i) $|\tau| \in \mathbb{N} \cup \{\infty\}$ is the length of a trace τ. (ii) $\tau_1 \cdot \tau_2$ is the concatenation of traces. (iii) $\tau[i, j)$ for $i, j \in \mathbb{N} \cup \{\infty\}$ is the subtrace beginning in the i-th state (inclusive) and ending before the j-th state. (Indices out of bounds are treated as $\tau[0, j)$ or $\tau[i, |\tau|)$, respectively.) (iv) $\tau[i]$ for $i < |\tau|$ is the state at position i in τ.

Definition 7 (Semantics of expressions). *Given a state s and a variable assignment β, the value $a^{s,\beta}$ of an expression a in a state s is the integer or boolean value resulting from interpreting program variables x by x^s, logical variables u by u^β, and using the standard interpretation for all functions and relations.*

Program expressions that do not contain logical variables are independent of β, and we write a^s instead of $a^{s,\beta}$. If a is a boolean expression, we write $s, \beta \models a$ resp. $s \models a$ to denote that $a^{s,\beta}$ resp. a^s is true.

As mentioned in Sect. 2, we consider assignments to global variables to be the only statements that lead to a new observable state. By specifying which variables are local and which are global, the user can thus determine which states are "interesting" and are to be included in a trace.

For the feasibility of proving DTL formulae, it is important that not too many irrelevant intermediate states are included in a trace because, if a formula such as $[\![\pi]\!]\Box\varphi$ is to be proven valid, intermediate states require sub-proofs showing that φ holds in each of them.

Definition 8 (Trace of a program). *Given an initial state s, the* trace of a program π, *denoted $trc(s,\pi)$, is defined by (the greatest fixpoint of):*

$$
\begin{array}{lcl}
trc(s,\epsilon) & = & \langle s\rangle \\
trc(s,\texttt{x = a; }\omega) & = & trc(s\{x\mapsto a^s\},\omega) \\
trc(s,\texttt{X = a; }\omega) & = & \langle s\rangle\cdot trc(s\{X\mapsto a^s\},\omega) \\
trc(s,\texttt{if (a) \{}\pi_1\texttt{\} else \{}\pi_2\texttt{\} }\omega) & = & \begin{cases} trc(s,\pi_1\ \omega) & \text{if } s\vDash a \\ trc(s,\pi_2\ \omega) & \text{if } s\nvDash a \end{cases} \\
trc(s,\texttt{while (a) \{}\pi\texttt{\} }\omega) & = & \begin{cases} trc(s,\pi\ \texttt{while (a) \{}\pi\texttt{\} }\omega) & \text{if } s\vDash a \\ trc(s,\omega) & \text{if } s\nvDash a \end{cases}
\end{array}
$$

where ϵ is the empty program and ω is a program.

We have now everything needed to define the semantics of DTL formulae in a straightforward way. The valuation of a formula is given w.r.t. a trace τ and a variable assignment β. This is expressed by the validity relation, denoted by $\vDash$.

Definition 9 (Semantics of formulae). *Let τ be a trace and β a variable assignment. The* validity relation *is the smallest relation satisfying the following.*

$$
\begin{array}{ll}
\tau,\beta\vDash a & \text{iff } a^{\tau[0],\beta} = \mathit{true} \\
 & \qquad \text{(in case } a \text{ is an expression, see Def. 7)} \\
\tau,\beta\vDash\neg\varphi & \text{iff } \tau,\beta\nvDash\varphi \\
\tau,\beta\vDash\varphi\wedge\psi & \text{iff } \tau,\beta\vDash\varphi \text{ and } \tau,\beta\vDash\psi \\
\tau,\beta\vDash\forall u.\varphi & \text{iff for every } d\in\mathbb{Z}\text{: } \tau,\beta\{u\mapsto d\}\vDash\varphi \\
\tau,\beta\vDash\Box\varphi & \text{iff } \tau[i,\infty),\beta\vDash\varphi \text{ for every } i\in[0,|\tau|) \\
\tau,\beta\vDash\varphi\,\mathbf{U}\,\psi & \text{iff } \tau[j,i),\beta\vDash\varphi \text{ and } \tau[i,\infty),\beta\vDash\psi \\
 & \qquad \text{for some } i\in[0,|\tau|) \text{ and all } j\in[0,i) \\
\tau,\beta\vDash\bullet\varphi & \text{iff } \tau[1,\infty),\beta\vDash\varphi \text{ or } |\tau|=1 \\
\tau,\beta\vDash\{x:=a\}\varphi & \text{iff } \tau\{x\mapsto a^{\tau[0]}\},\beta\vDash\varphi \\
\tau,\beta\vDash[\![\pi]\!]\varphi & \text{iff } trc(\tau[0],\pi),\beta\vDash\varphi
\end{array}
$$

A formula φ is valid *if $\tau,\beta\vDash\varphi$ for all τ and all β.*

4 A Sequent Calculus for DTL

In this section, we present a sequent calculus for DTL, which we call $\mathcal{C}_{\mathrm{DTL}}$. It is sound and relatively complete, i.e., complete up to the handling of arithmetic (see Sect. 5). The calculus consists of the following rule classes:

Classical Logic Rules. These rules simplify formulae whose top-level operator is a quantifier or a propositional operator.
Simplification and Normalization Rules. Rules for simplifying formulae of the form $\mathcal{U}[\![\pi]\!]\varphi$, where the top-level operator in φ is not temporal.
Rules for Temporal Operators. Rules that apply to formulae $\mathcal{U}[\![\pi]\!]\varphi$ with a top-level temporal operator in φ, and that do not change the program π.
Program Rules. Rules that apply to formulae of the form $\mathcal{U}[\![\pi]\!]\varphi$, and that analyze and/or simplify the program π. Not surprisingly, this class has the most complex rules, including invariant rules for loops.
Rules for Data Structures. Since our focus in this paper is not on how to handle arithmetics, we use oracle rules for arithmetics.
Other Rules. This category includes the closure and the cut rule.

Most rules of the calculus are analytic and therefore can be applied automatically. The rules that require user interaction are: (a) the rules for handling while loops (where a loop invariant has to be provided), (b) the cut rule (where the right case distinction has to be used), and (c) the quantifier rules (where the right instantiation has to be found). Traces are uniquely determined by symbolic program executions of the deterministic programing language. The general idea behind our calculus is to explore a trace until it terminates or reaches a fixpoint (induced by a non-terminating loop). Thus, proofs usually consist of alternating applications of temporal logic rules (which decompose trace formulae, e.g., $\Box\varphi$ to $\bullet\Box\varphi \wedge \varphi$) and program rules (which let us step forward in the trace). Those steps are explicitly given through assignments in the program.

In the rule schemata, Γ, Δ denote arbitrary, possibly empty multi-sets of formulae, φ, ψ denote arbitrary formulae, $\mathcal{U}$ stands for a (possibly empty) sequence of updates, π, ω for programs, γ is a state formula, `x` and `X` are local and global program variables, n and u are logical variables, a is an expression of type integer, and b is an expression of type boolean.

As usual, the schematic sequents above the horizontal line in a schema are its *premisses* and the single schematic sequent below the horizontal line is its *conclusion*. Note, that in practice the rules are applied from bottom to top. Proof construction starts with the original proof obligation at the bottom. Therefore, if a constraint is attached to a rule that requires a variable to be "new," it has to be new w.r.t. the conclusion.

Definition 10 (Soundness, derivability).

1. *A sequent $\Gamma \vdash \Delta$ is* valid *(in state s and under variable assignment β) if and only if the formula $\bigwedge_{\gamma\in\Gamma}\gamma \rightarrow \bigvee_{\delta\in\Delta}\delta$ is valid (w.r.t. s, β).*
2. *A rule* $\dfrac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma_0 \vdash \Delta_0}$ *is* sound *if, for all valid instances of premisses $\Gamma_i \vdash \Delta_i$, also the instance of $\Gamma_0 \vdash \Delta_0$ is valid.*
3. *A sequent is* derivable *(with $\mathcal{C}_{\mathrm{DTL}}$) if it is an instance of the conclusion of a rule schema and all corresponding instances of the premisses of that rule schema are derivable sequents. In particular, all sequents are derivable that are instances of the conclusion of a rule that has no premisses (rules R22, R31, and R33).*

Table 1. Rules for quantifiers, propositional operators, and state updates. In rule R5, the substitution needs to be admissible; rule R6 introduces a fresh variable u'. Rules R7 and R8 make use of weak substitution (Def. 12).

$$\frac{\Gamma \vdash \varphi, \Delta}{\Gamma, \neg\varphi \vdash \Delta}\ \mathsf{R1} \qquad \frac{\Gamma, \varphi \vdash \Delta}{\Gamma \vdash \neg\varphi, \Delta}\ \mathsf{R2}$$

$$\frac{\Gamma, \varphi, \psi \vdash \Delta}{\Gamma, \varphi \wedge \psi \vdash \Delta}\ \mathsf{R3} \qquad \frac{\Gamma \vdash \varphi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \wedge \psi, \Delta}\ \mathsf{R4}$$

$$\frac{\Gamma, \varphi[u/a], \forall u.\varphi \vdash \Delta}{\Gamma, \forall u.\varphi \vdash \Delta}\ \mathsf{R5} \qquad \frac{\Gamma \vdash \varphi[u/u'], \Delta}{\Gamma \vdash \forall u.\varphi, \Delta}\ \mathsf{R6}$$

$$\frac{\Gamma, \mathcal{U}\varphi[x\sharp a] \vdash \Delta}{\Gamma, \mathcal{U}\{x := a\}\varphi \vdash \Delta}\ \mathsf{R7} \qquad \frac{\Gamma \vdash \mathcal{U}\varphi[x\sharp a], \Delta}{\Gamma \vdash \mathcal{U}\{x := a\}\varphi, \Delta}\ \mathsf{R8}$$

4.1 Classical Logic and Update Rules

The rules for quantifiers, propositional operators, and updates are shown in Table 1. Note that the expressions that are used to instantiate universal quantifiers in rule R5 must be chosen in such a way that the substitution is admissible:

Definition 11 (Admissible substitution). *A substitution u/a of a logical variable $u \in V$ with an expression a is* admissible *w.r.t. a formula φ if there is no variable v in a such that u is free in φ and, after replacing a for some free occurrence of u in φ, the occurrence of v in a is* (i) *bound by a quantifier in $\varphi[u/a]$ (in case v is a logical variable) or is* (ii) *in the scope of a program modality $[\![\pi]\!]$ that contains an assignment to v (in case v is a program variable).*

For example, using `X` to instantiate the universal quantifier in the DTL formula $\forall u.(u \doteq 0 \rightarrow [\![\texttt{X = 1;}]\!]\Box u \doteq 0)$ is not admissible. Indeed the result would be incorrect as the original formula is valid while $X \doteq 0 \rightarrow [\![\texttt{X = 1;}]\!]\Box X \doteq 0$ is not even satisfiable. In order to deal with updates, we introduce the notion of *weak substitutions*, which avoid such clashes by definition.

Definition 12 (Weak substitution). *For a state formula φ and an update $\{x := a\}$ define the formula $\varphi[x\sharp a]$ according to the following schema:* (i) *if φ is an expression, then $\varphi[x\sharp a] = \varphi[x/a]$,* (ii) *if φ begins with an update or a program modality, then $\varphi[x\sharp a] = \{x := a\}\varphi$,* (iii) *if φ is a propositional junction, then the weak substitution is propagated, e.g., $(\varphi_1 \wedge \varphi_2)[x\sharp a] = \varphi_1[x\sharp a] \wedge \varphi_2[x\sharp a]$,* (iv) *if φ begins with a quantifier, then the weak substitution is propagated (possibly under renaming the bound variable so that it does not occur in a).*

4.2 Simplification and Normalization Rules

As said above, our calculus contains simplification rules that apply to formulae of the form $\mathcal{U}[\![\pi]\!]\varphi$, where the top-level operator in φ is not temporal. They are shown in Table 2. In particular, they include normalization rules which deal with negated trace formulae through replacement by the respective dual formula.

Table 2. Simplification and normalization rules. In rule R16, γ is a state formula. Rule R17 introduces a fresh variable u'; in rule R18, the substitution needs to be admissible.

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\varphi, \mathcal{U}[\![\pi]\!]\psi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!](\varphi \vee \psi), \Delta}\ \mathsf{R9} \qquad \frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\varphi, \Delta \quad \Gamma \vdash \mathcal{U}[\![\pi]\!]\psi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!](\varphi \wedge \psi), \Delta}\ \mathsf{R10}$$

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\neg\varphi, \Delta}{\Gamma \vdash \neg\mathcal{U}[\![\pi]\!]\varphi, \Delta}\ \mathsf{R11} \qquad \frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\Box\neg\psi,\ \mathcal{U}[\![\pi]\!](\neg\psi\ \mathbf{U}\ (\neg\varphi \wedge \neg\psi)), \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\neg(\varphi\ \mathbf{U}\ \psi), \Delta}\ \mathsf{R12}$$

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\neg\varphi, \Delta}{\Gamma, \mathcal{U}[\![\pi]\!]\varphi \vdash \Delta}\ \mathsf{R13} \qquad \frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\varphi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\neg\neg\varphi, \Delta}\ \mathsf{R14}$$

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\circ\neg\varphi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\neg\bullet\varphi, \Delta}\ \mathsf{R15} \qquad \frac{\Gamma \vdash \mathcal{U}\gamma, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\gamma, \Delta}\ \mathsf{R16}$$

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\varphi[u/u'], \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\forall u.\varphi, \Delta}\ \mathsf{R17} \qquad \frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\varphi[u/a], \mathcal{U}[\![\pi]\!]\exists u.\varphi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\exists u.\varphi, \Delta}\ \mathsf{R18}$$

Rule R12 for negated until avoids introducing the dual $\mathbf{R}$ into the sequent. Therefore, no rules for $\mathbf{R}$ are required in the calculus. Soundness of R12 follows from the well-known equivalence $\varphi\,\mathbf{R}\,\psi \leftrightarrow \psi\,\mathbf{W}\,(\varphi \wedge \psi)$ in LTL and the definitions of $\mathbf{R}$ and $\mathbf{W}$, which applies to finite traces as well (cf., e.g., [2]).

Since (for conciseness of the calculus) we only include program and temporal logic rules for the right-hand side of a sequent, we need rule R13 that allows to move a formula with a modality from the left of a sequence to the right.

In case φ is a state formula, rule R16 can be used to remove the program modality (as a state formula is evaluated in the initial state of a trace). Further simplification rules are applied to split formulae such as $[\![\pi]\!](\Box\varphi \wedge \psi)$.

4.3 Rules for Temporal Operators

Table 3 shows the rules that handle temporal operators without changing the program. Rules R19 to R21 "unwind" temporal formulae by splitting them into a "future" part and a "present" part. Rules R22 and R23 handle the case of an empty program (i.e., empty remaining trace) for weak and strong next, respectively. Rule R22 also closes a proof branch.

Table 3. Rules for handling temporal operators

$$\frac{\Gamma \vdash \mathcal{U}([\![\pi]\!]\circ(\varphi\ \mathbf{U}\ \psi) \wedge [\![\pi]\!]\varphi), \mathcal{U}[\![\pi]\!]\psi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\varphi\ \mathbf{U}\ \psi, \Delta}\ \mathsf{R19} \qquad \frac{\Gamma \vdash \mathcal{U}([\![\pi]\!]\bullet\Box\varphi \wedge [\![\pi]\!]\varphi), \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\Box\varphi, \Delta}\ \mathsf{R20}$$

$$\frac{\Gamma \vdash \mathcal{U}[\![\pi]\!]\circ\Diamond\varphi, \mathcal{U}[\![\pi]\!]\varphi, \Delta}{\Gamma \vdash \mathcal{U}[\![\pi]\!]\Diamond\varphi, \Delta}\ \mathsf{R21} \qquad \frac{}{\Gamma \vdash \mathcal{U}[\![]\!]\bullet\varphi, \Delta}\ \mathsf{R22} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \mathcal{U}[\![]\!]\circ\varphi, \Delta}\ \mathsf{R23}$$

4.4 Program Rules

The program rules are shown in Table 4. Assignments to local and global variables are handled by the rules R24 and R26, respectively. The former can be applied on any formula φ, while the latter one, which handles assignments to global variables, steps to the next state and consumes a (weak or strong) next operator.

Table 4. Program rules. The schematic symbol $\overset{\bullet}{\circ}$ stands for $\bullet$ or $\circ$.

$$\frac{\Gamma \vdash \mathcal{U}\{x := a\}[\![\omega]\!]\varphi, \Delta}{\Gamma \vdash \mathcal{U}[\![\texttt{x = a; } \omega]\!]\varphi, \Delta}\ \text{R24} \qquad \frac{\Gamma,\ \mathcal{U}b \vdash \mathcal{U}[\![\pi_1\ \omega]\!]\varphi, \Delta \qquad \Gamma,\ \mathcal{U}\neg b \vdash \mathcal{U}[\![\pi_2\ \omega]\!]\varphi, \Delta}{\Gamma \vdash \mathcal{U}[\![\texttt{if (b) \{}\pi_1\texttt{\} else \{}\pi_2\texttt{\}}\ \omega]\!]\varphi, \Delta}\ \text{R25}$$

$$\frac{\Gamma \vdash \mathcal{U}\{X := a\}[\![\omega]\!]\varphi, \Delta}{\Gamma \vdash \mathcal{U}[\![\texttt{X = a; } \omega]\!]\overset{\bullet}{\circ}\varphi, \Delta}\ \text{R26} \qquad \frac{\Gamma \vdash \mathcal{U}[\![\texttt{if (b)\{}\pi\ \texttt{while (b) \{}\pi\texttt{\}\} else \{\}}\ \omega]\!]\varphi, \Delta}{\Gamma \vdash \mathcal{U}[\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\varphi, \Delta}\ \text{R27}$$

An `if` statement is handled by splitting the formula in two parts, each containing the alternative program and the remaining program code as shown in rule R25. Similarly, loops can be handled by unwinding, as shown in rule R27. In the case in which the loop condition holds, the loop body is symbolically executed and then again the whole loop. In the second case where the loop condition does not hold, the loop is simply skipped. However, the number of loop iterations may not be known in advance, or the loop may not even terminate. In those cases, we need invariants.

Invariant rules are an established technique for handling loops in calculi for program logics. The basic idea is to have a state formula γ (the invariant) which holds in all states before and – if it terminates – after an execution of the loop body. If we can show that preservation, it only remains to show that φ holds on the remaining trace. The rules are shown in Table 5.

For a trace formula of the shape $\Box\varphi$, the four premisses of R28 intuitively state that (i) γ holds in the beginning; (ii) it is preserved by each loop iteration (i.e., it actually is an invariant), here a possible post-π state is characterized

Table 5. Invariant rules

$$\frac{\Gamma \vdash \mathcal{U}\gamma, \Delta \qquad \gamma, b \vdash [\![\pi]\!]\Box(\bullet \mathit{false} \rightarrow \gamma) \qquad \gamma \vdash b, [\![\omega]\!]\Box\varphi \qquad \gamma, b \vdash [\![\pi \mid \texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\varphi}{\Gamma \vdash \mathcal{U}[\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Box\varphi, \Delta}\ \text{R28}$$

$$\frac{\Gamma \vdash \exists u.(u \geq 0 \wedge \mathcal{U}\mathcal{V}_u\gamma), \Delta \qquad n \geq 0 \vdash \mathcal{V}_{n+1}(\gamma \rightarrow (b \wedge [\![\pi]\!]\Diamond(\bullet \mathit{false} \wedge \mathcal{V}_n\gamma))) \qquad \vdash \mathcal{V}_0(\gamma \rightarrow [\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Diamond\varphi)}{\Gamma \vdash \mathcal{U}[\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\Diamond\varphi, \Delta}\ \text{R29}$$

$$\frac{\Gamma \vdash \exists u.(u \geq 0 \wedge \mathcal{U}\mathcal{V}_u\gamma), \Delta \qquad n \geq 0 \vdash \mathcal{V}_{n+1}(\gamma \rightarrow (b \wedge [\![\pi]\!]\Diamond(\bullet \mathit{false} \wedge \mathcal{V}_n\gamma))) \qquad \vdash \mathcal{V}_0(\gamma \rightarrow [\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\varphi_1\ \mathbf{U}\ \varphi_2) \qquad n > 0 \vdash \mathcal{V}_n(\gamma \rightarrow [\![\pi \mid \texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\varphi_1)}{\Gamma \vdash \mathcal{U}[\![\texttt{while (b) \{}\pi\texttt{\}}\ \omega]\!]\varphi_1\ \mathbf{U}\ \varphi_2, \Delta}\ \text{R30}$$

by the temporal formula $\bullet \mathit{false}$; (iii) if the loop terminates, indicated by the negated loop condition b, then $\Box\varphi$ holds on the remaining trace; and (iv) for every loop iteration, φ holds throughout, i.e., for the remaining trace from every state during loop iterations. As an invariant abstracts from concrete loop iterations, the context Γ, Δ must be discarded in the all but the first premiss.

Note that – in contrast to invariant rules in state-based dynamic logic – it is not sound in premiss (iv), to decompose the program trace and to only regard the subtrace induced by π in isolation, i.e., just proving $[\![\pi]\!]\Box\varphi$ is not sound. As an example, consider the formula $[\![\texttt{while (X>0) \{X = X-1;\}}]\!]\Box\bullet\bullet \mathit{false}$, which is not valid, but the formula $[\![\texttt{X = X-1;}]\!]\Box\bullet\bullet \mathit{false}$, containing the loop body, obviously is. This means for a sound rule, that we have to consider the remaining trace as well. However, we are only interested in those traces which begin in the subtrace induced by the loop body π.

For this reason, we introduced another, two-place program modality: $[\![\pi \mid \omega]\!]\varphi$ means that for any state in the subtrace induced by π, trace formula φ holds for the remaining trace including ω. More formally, we define $[\![\pi \mid \omega]\!]\varphi$ as a shorthand for $[\![\texttt{x = 0;}\ \pi\ \texttt{x = 1;}\omega]\!](\varphi\, \mathbf{W}\, x \doteq 1)$ where local program variable x does not occur in π, ω, or φ. Even though the resulting formula is syntactically longer here, it is easier to prove in the sense that there are fewer states in which φ has to hold.

In the case of R29 ("diamond") and R30 ("until"), the invariant is accompanied by a sequence of updates $\mathcal{V}_u$ with an integer expression u, which describes the progress made through each loop iteration. The general shape of $\mathcal{V}_u$ is $\{x_1 := f_1(u)\} \cdots \{x_k := f_k(u)\}$ where $x_1, \ldots, x_k$ are variables appearing in γ and $f_1, \ldots, f_k$ are functions. The intuition behind it is that $\mathcal{V}_0\gamma$ describes either a state in which the loop terminates immediately or a fixpoint of the loop. Such a state must be reached in a finite number of iterations, which is guaranteed since n is decreasing in every iteration. For this reason, premiss (ii) requires executions of the loop body to terminate. In Rule R30, there is a fourth premiss stating that φ_1 holds throughout the loop body for every iteration where $n > 0$.

4.5 Rules for Data Structures

Our calculus is basically independent of the domain of computation resp. data structures that are used. We therefore abstract from the problem of handling the data structure(s) and just assume that an oracle is available that can decide the validity of non-temporal formulae in the domain of computation (note that the oracle only decides pure first-order formulae). In the case of arithmetic, the oracle is represented by rule R31 in Table 6.

Of course, the non-temporal formulae that are valid in arithmetic are not even enumerable. Therefore, in practice, the oracle can only be approximated, and rule R31 must be replaced by a rule (or set of rules) for computing resp. enumerating a *subset* of all valid non-temporal formulae (in particular, these rules must include equality handling). This is not harmful to "practical completeness." Rule sets for arithmetic are available, which – as experience shows – allow to derive all valid non-temporal formulae that occur during the verification of

Table 6. Oracle rules and induction rule for handling arithmetic (n is fresh)

if $\bigwedge \Gamma \to \bigvee \Delta$ is a valid non-temporal formula: $\dfrac{}{\Gamma \vdash \Delta}$ R31

$$\frac{\Gamma \vdash \varphi(0), \Delta \qquad \Gamma, \varphi(u) \vdash \varphi(u+1), \Delta}{\Gamma \vdash \forall u.\varphi(u), \Delta} \text{ R32}$$

Table 7. The closure and the cut rule

$$\frac{}{\Gamma, \varphi \vdash \varphi, \Delta} \text{ R33} \qquad \frac{\Gamma, \varphi \vdash \Delta \qquad \Gamma \vdash \varphi, \Delta}{\Gamma \vdash \Delta} \text{ R34}$$

actual programs. Using powerful SMT solvers, this can be done fully automatically in many cases. Typically, an approximation of the computation domain oracle contains a rule for structural induction. In the case of arithmetic, that is rule R32. This rule, however, not only applies to non-temporal formulae but also to DTL formulae containing programs.

The remaining rules, which are shown in Table 7, are the cut rule R34 (with an arbitrary cut formula φ) and the closure rule R33 which closes a proof branch.

5 Soundness and Completeness

Soundness of the calculus $\mathcal{C}_{\mathrm{DTL}}$ (Corollary 1) is based on the following theorem, which states that all rules preserve validity of the derived sequents.

Theorem 1. *For all rule schemata of the calculus $\mathcal{C}_{\mathrm{DTL}}$, R1 to R34, the following holds: If all premisses of a rule schema instance are valid sequents, then its conclusion is a valid sequent.*

Corollary 1. *If a sequent $\Gamma \vdash \Delta$ is derivable with the calculus $\mathcal{C}_{\mathrm{DTL}}$, then it is valid, i.e., $\bigwedge \Gamma \to \bigvee \Delta$ is a valid formula.*

Proving Theorem 1 is not difficult. The proof is, however, quite large as soundness has to be shown separately for each rule; the proof is given in [4, App. A].

The calculus $\mathcal{C}_{\mathrm{DTL}}$ is *relatively* complete; that is, it is complete up to the handling of the domain of computation (the data structures). It is complete if an oracle rule for the domain is available – in our case the oracle rule for arithmetic, R31. If the domain is extended with other data types, $\mathcal{C}_{\mathrm{DTL}}$ remains relatively complete; and it is still complete if rules for handling the extended domain of computation are added.

Theorem 2. *If a sequent is valid, then it is derivable with $\mathcal{C}_{\mathrm{DTL}}$.*

Corollary 2. *If φ is a valid DTL formula, then the sequent $\vdash \varphi$ is derivable.*

Due to space restrictions, the proof of Theorem 2, which is quite complex, cannot be given here. The basic idea of the proof is the same as that used by Harel [8] to prove relative completeness of his sequent calculus for first-order DL. An extensive proof sketch can be found in [4, App. B]. The following lemma is central to the completeness proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \{X := 5\}[\![]\circ\Diamond X \geq 4,\ 5 \geq 4,\ [\![\texttt{X=5;}]\!]X \geq 4}\,\text{R31}}{\vdash \{X := 5\}[\![]\circ\Diamond X \geq 4,\ \{X := 5\}X \geq 4,\ [\![\texttt{X=5;}]\!]X \geq 4}\,\text{R8}}{\vdash \{X := 5\}[\![]\circ\Diamond X \geq 4,\ \{X := 5\}[\![]X \geq 4,\ [\![\texttt{X=5;}]\!]X \geq 4}\,\text{R16}}{\vdash \{X := 5\}[\![]\Diamond X \geq 4,\ [\![\texttt{X=5;}]\!]X \geq 4}\,\text{R21}}{\vdash [\![\texttt{X=5;}]\!]\circ\Diamond X \geq 4,\ [\![\texttt{X=5;}]\!]X \geq 4}\,\text{R26}}{\vdash [\![\texttt{X=5;}]\!](\circ\Diamond X \geq 4 \vee X \geq 4)}\,\text{R9}}{\vdash [\![\texttt{X=5;}]\!]\Diamond X \geq 4}\,\text{R21}$$

Fig. 1. Example proof tree (rules focus on the solid black formulae)

Lemma 1. *For every DTL formula φ_{DTL} there is an (arithmetical) non-temporal first-order formula φ_{FOL} that is logically equivalent to φ_{DTL}, i.e., for all traces τ and variable assignments β:*

$$\tau, \beta \vDash \varphi_{DTL} \quad \textit{iff} \quad \tau, \beta \vDash \varphi_{FOL}\ .$$

The above lemma states that DTL is not more expressive than first-order arithmetic. This holds as arithmetic – our domain of computation – is expressive enough to encode the behaviour of programs. In particular, using Gödelization, arithmetic allows to encode program states (i.e., the values of all the variables occurring in a program) and finite (sub-)traces into a single number. Further it is then possible to construct, for every DTL formula ψ, state s, program π, and $n \in \mathbb{N}$, a FOL formula $\varphi_{\psi,s,\pi,n}$ encoding that $trc(s,\pi)[n,\infty) \vDash \psi$.

Note that Lemma 1 states a property of the logic DTL that is independent of any calculus. It implies that a DTL formula could be decided by constructing an equivalent non-temporal formula and then invoking the computation domain oracle – if such an oracle were actually available. But even with a good approximation of an arithmetic oracle, that is not practical (the non-temporal first-order formula would be too complex to prove automatically or interactively). And, indeed, the calculus $\mathcal{C}_{\mathrm{DTL}}$ does not work that way.

The (relative) completeness of $\mathcal{C}_{\mathrm{DTL}}$ requires an expressive computation domain and is lost if a simpler domain and less expressive data structures are used. The reason is that in a simpler domain it may not be possible to express the required invariants for all possible while loops.

6 Conclusions and Further Directions

In this paper, we have defined the logic DTL, which stems from a novel combination of dynamic logic and first-order temporal logic. In contrast to [6,10], there is no restriction on the shape of trace formulae. Through this, we have got an expressive logic allowing to describe complex temporal properties of programs. An example proof can be found in Figure 1. Of course, this is a fairly simple program and trace property, but it already requires some proof steps. More elaborate examples (e.g., including proof splits) cannot be given in this paper due to limited space.

One major aim of this work is to express information flow properties in a concurrent setting. In current work in progress, we have sketched an idea how to reason about possible information flows throughout program execution. We still regard only sequential executions of sub-programs (i.e., threads), but execution traces instead of initial and final states. The rationale behind this is that an attacker may be in control of another thread running on the same memory and thus may read variables at any time. For absence of information flow, we show that traces beginning in states which only differ in the values of secret variables are bisimilar in public observations. In earlier work, the information flow policy of non-interference for sequential programs is expressed through self-composition of dynamic logic formulae [12]. This basic idea can be combined with declassification, i.e., the controlled release of information, under temporal constraints, which means to specify *when* information may be released.

State-based dynamic logics, both for deterministic and indeterministic languages, have the well-known property of compositionality. For example, the formulae $[\pi\ \omega]\varphi$ and $[\pi][\omega]\varphi$ are logically equivalent. This is important since program complexity imports much to the overall complexity of a DL formula. This does not apply to our situation as traces may not be decomposed in general. For purposes like loop invariants (see Table 5), however, program decompositions are indispensable. This has lead us to the auxiliary notation $[\![\pi \mid \omega]\!]\varphi$, which talks about all traces beginning in π but extending into ω. Another possibility to make proofs more feasible would be to introduce additional rules for special, commonly used patterns of trace formulae – such as $\Box\Diamond\gamma$ where γ is a state formula – for which we know that decompositions are sound.

The sequent calculus $\mathcal{C}_{\mathrm{DTL}}$ here has been prototypically implemented in the current development version of the interactive KeY prover. Instead of the simple toy language introduced in this paper, the implemented calculus works on actual Java programs. The efforts so far suggest that most program rules can be adapted straight away from the present rules for the $[\cdot]$ modality.

Acknowledgement. The authors would like to thank Mattias Ulbrich, Andreas Wagner, and the anonymous reviewers for their helpful and supporting comments.

References

1. Abadi, M., Manna, Z.: Nonclausal deduction in first-order temporal logic. Journal of the ACM 37(2), 279–317 (1990)
2. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010)
3. Beckert, B.: A dynamic logic for Java Card. In: Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France, pp. 111–119 (2000)
4. Beckert, B., Bruns, D.: Dynamic trace logic: Definition and proofs. Tech. Rep. 2012-10, Karlsruhe Institute of Technology, Department of Computer Science (2012), revised version available at
`http://formal.iti.kit.edu/~bruns/papers/trace-tr.pdf`

5. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
6. Beckert, B., Schlager, S.: A sequent calculus for first-order dynamic logic with trace modalities. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 626–641. Springer, Heidelberg (2001)
7. Goré, R.: Tableau methods for modal and temporal logics. In: D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J. (eds.) Handbook of Tableau Methods, pp. 297–396. Kluwer Academic Publishers, Dordrecht (1999)
8. Harel, D.: Dynamic logic. In: Gabbay, D., Guenther, F. (eds.) Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic, pp. 497–604. D. Reidel Publishing Co., Dordrecht (1984)
9. Moszkowski, B.: A temporal logic for multilevel reasoning about hardware. IEEE Computer 18(2) (February 1985)
10. Platzer, A.: A temporal dynamic logic for verifying hybrid system invariants. In: Artemov, S., Nerode, A. (eds.) LFCS 2007. LNCS, vol. 4514, pp. 457–471. Springer, Heidelberg (2007)
11. Reynolds, M., Dixon, C.: Theorem-proving for discrete temporal logic. In: Fisher, D.M., Gabbay, Vila, L. (eds.) Handbook of Temporal Reasoning in Artificial Intelligence. Elsevier Science (2005)
12. Scheben, C., Schmitt, P.H.: Verification of information flow properties of JAVA programs without approximations. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 232–249. Springer, Heidelberg (2012)
13. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In: Combi, C., Leucker, M., Wolter, F. (eds.) Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011, pp. 99–106. IEEE (2011)
14. Thums, A., Schellhorn, G., Ortmeier, F., Reif, W.: Interactive verification of statecharts. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 355–373. Springer, Heidelberg (2004)
15. Wolper, P.: The tableau method for temporal logic: An overview. Logique et Analyse 28(110-111), 119–136 (1985)

Temporalizing Ontology-Based Data Access*

Franz Baader, Stefan Borgwardt, and Marcel Lippmann

TU Dresden, Germany
{baader,stefborg,lippmann}@tcs.inf.tu-dresden.de

Abstract. Ontology-based data access (OBDA) generalizes query answering in databases towards deduction since (i) the fact base is not assumed to contain complete knowledge (i.e., there is no closed world assumption), and (ii) the interpretation of the predicates occurring in the queries is constrained by axioms of an ontology. OBDA has been investigated in detail for the case where the ontology is expressed by an appropriate Description Logic (DL) and the queries are conjunctive queries. Motivated by situation awareness applications, we investigate an extension of OBDA to the temporal case. As query language we consider an extension of the well-known propositional temporal logic LTL where conjunctive queries can occur in place of propositional variables, and as ontology language we use the prototypical expressive DL $\mathcal{ALC}$. For the resulting instance of temporalized OBDA, we investigate both data complexity and combined complexity of the query entailment problem.

1 Introduction

Situation awareness tools [2,12] try to help the user to detect certain situations within a running system. Here "system" is seen in a broad sense: it may be a computer system, air traffic observed by radar, or a patient in an intensive care unit. From an abstract point of view, the system is observed by certain "sensors" (e.g., heart-rate and blood pressure monitors for a patient), and the results of sensing are stored in a fact base. Based on the information available in the fact base, the situation awareness tool is supposed to detect certain predefined situations (e.g., heart-rate very high and blood pressure low), which require a reaction (e.g., fetch a doctor or give medication).

In a simple setting, one could realize such a tool by using standard database techniques: the information obtained from the sensors is stored in a relational database, and the situations to be recognized are specified by queries in an appropriate query language (e.g., conjunctive queries [1]). However, in general we cannot assume that the sensors provide us with a complete description of the current state of the system, and thus the closed world assumption (CWA) employed by database systems (where facts not occurring in the database are assumed to be false) is not appropriate (since there may be facts for which it is not known whether they are true or false). In addition, though one usually does not have a complete specification of the working of the system (e.g., a

* Partially supported by DFG SFB 912 (HAEC) and GRK 1763 (QuantLA).

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 330–344, 2013.

complete biological model of a human patient), one has some knowledge about how the system works. This knowledge can be used to formulate constraints on the interpretation of the predicates used in the queries, which may cause more answers to be found.

Ontology-based data access [11,17] addresses these requirements. The fact base is viewed to be a Description Logic ABox (which is not interpreted with the CWA), and an ontology, also formulated in an appropriate DL, constrains the interpretations of unary and binary predicates, called concepts and roles in the DL community. As an example, assume that the ABox $\mathcal{A}$ contains the following assertions about the patient Bob:

$$\begin{array}{c} \mathit{systolic\_pressure}(\mathit{BOB}, \mathit{P1}), \quad \mathit{High\_pressure}(P1), \\ \mathit{history}(\mathit{BOB}, H1), \quad \mathit{Hypertension}(H1), \quad \mathit{Male}(\mathit{BOB}) \end{array}$$

which say that Bob has high blood pressure (obtained from sensor data), and is male and has a history of hypertension (obtained from the patient records). In addition, we have an ontology that says that patients with high blood pressure have hypertension and that patients that currently have hypertension and also have a history of hypertension are at risk for a heart attack:

$$\begin{array}{c} \exists \mathit{systolic\_pressure}.\mathit{High\_pressure} \sqsubseteq \exists \mathit{finding}.\mathit{Hypertension} \\ \exists \mathit{finding}.\mathit{Hypertension} \sqcap \exists \mathit{history}.\mathit{Hypertension} \sqsubseteq \exists \mathit{risk}.\mathit{Myocardial\_infarction} \end{array}$$

The situation we want to recognize for a given patient x is whether this patient is a male person that is at risk for a heart attack. This situation can be described by the conjunctive query $\exists y.\mathit{risk}(x, y) \wedge \mathit{Myocardial\_infarction}(y) \wedge \mathit{Male}(x)$. Given the information in the ABox and the axioms in the ontology, we can derive that Bob satisfies this query, i.e., he is a *certain answer* of the query. Obviously, without the ontology this answer could not be derived.

The complexity of OBDA, i.e., the complexity of checking whether a given tuple of individuals is a certain answer of a conjunctive query in an ABox w.r.t. an ontology, has been investigated in detail for cases where the ontology is expressed in an appropriate DL and the query is a conjunctive query. One can either consider the *combined complexity*, which is measured in the size of the whole input (consisting of the query, the ontology, and the ABox), or the *data complexity*, which is measured in the size of the ABox only (i.e., the query and the ontology are assumed to be of constant size). The underlying assumption is that query and ontology are usually relatively small, whereas the size of the data may be huge. In the database setting (where there is no ontology and CWA is used), answering conjunctive queries is NP-complete w.r.t. combined complexity and in AC^0 w.r.t. data complexity [8,1]. For expressive DLs, the complexity of checking certain answers is considerably higher. For instance, for the well-known DL $\mathcal{ALC}$, OBDA is ExpTime-complete w.r.t. combined complexity and co-NP-complete w.r.t. data complexity [7,13,6]. For this reason, more light-weight DLs have been developed, for which the data complexity of OBDA is still in AC^0 and for which computing certain answers can be reduced to answering conjunctive queries in the database setting [5].

Unfortunately, OBDA as described until now is not sufficient to achieve situation awareness. The reason is that the situations we want to recognize may depend on states of the system at different time points. For example, assume that we want to find male patients that have a history of hypertension, i.e., patients that are male and at some previous time point had hypertension.[1] In order to express this kind of temporal queries, we propose to extend the well-known propositional temporal logic LTL [16] by allowing the use of conjunctive queries in place of propositional variables. For example, male patients with a history of hypertension can then be described by the query

$$\mathit{Male}(x) \wedge \bigcirc^- \Diamond^- (\exists y.\mathit{finding}(x, y) \wedge \mathit{Hypertension}(y)),$$

where $\bigcirc^-$ stands for "previous" and $\Diamond^-$ stands for "sometime in the past." The query language obtained this way extends the temporal description logic $\mathcal{ALC}$-LTL introduced and investigated in [4]. In $\mathcal{ALC}$-LTL, only concept and role assertions (i.e., very restricted conjunctive queries without variables and existential quantification) can be used in place of propositional variables. As in [4], we also consider rigid concepts and roles, i.e., concepts and roles whose interpretation does not change over time. For example, we may want to assume that the concept *Male* is rigid, and thus a patient that is male now also has been male in the past and will stay male in the future.

Our overall setting for recognizing situations will thus be the following. In addition to a global ontology $\mathcal{T}$ (which describes properties of the system that hold at every time point, using the DL $\mathcal{ALC}$), we have a sequence of ABoxes $\mathcal{A}_0, \mathcal{A}_1, \ldots \mathcal{A}_n$, which (incompletely) describe the states of the system at the previous time points $0, 1, \ldots, n-1$ and the current time point n. The situation to be recognized is expressed by a temporal conjunctive query, as introduced above, which is evaluated w.r.t. the current time point n. We will investigate both the combined and the data complexity of this temporal extension of OBDA in three different settings: (i) both concepts and roles may be rigid; (ii) only concepts may be rigid; and (iii) neither concepts nor roles are allowed to be rigid. For the combined complexity, the obtained complexity results are identical to the ones for $\mathcal{ALC}$-LTL, though the upper bounds are considerably harder to show. For the data complexity, the results for the settings (ii) and (iii) coincides with the one for atemporal OBDA (co-NP-complete). For the setting (i), we can show that the data complexity is in ExpTime (in contrast to 2-ExpTime-completeness for the combined complexity), but we do not have a matching lower bound.

The details of the proofs can be found in the accompanying technical report [3].

2 Preliminaries

While in principle our temporal query language can be parameterized with any DL, in this paper we focus on $\mathcal{ALC}$ [20] as a prototypical expressive DL.

[1] Whereas in the previous example we have assumed that a history of hypertension was explicitly noted in the patient records, we now want to derive this information from previously stored information about blood pressure, etc.

Definition 2.1 (syntax of $\mathcal{ALC}$). *Let N_C, N_R, and N_I, respectively, be non-empty, pairwise disjoint sets of* concept names, role names, *and* individual names. *The set of* concept descriptions *(or* concepts*) is the smallest set such that all concept names $A \in N_C$ are concepts, and if C, D are concepts and $r \in N_R$, then $\neg C$ (negation), $C \sqcap D$ (conjunction), and $\exists r.C$ (existential restriction) are also concepts.*

A general concept inclusion (GCI) *is of the form $C \sqsubseteq D$, where C, D are concepts, and an* assertion *is of the form $C(a)$ or $r(a, b)$, where C is a concept, $r \in N_R$, and $a, b \in N_I$. We call both GCIs and assertions* axioms. *A* TBox *(or* ontology*) is a finite set of GCIs and an* ABox *is a finite set of assertions.*

The semantics of $\mathcal{ALC}$ is defined in a model-theoretic way.

Definition 2.2 (semantics of $\mathcal{ALC}$). *An interpretation is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty set (called* domain*), and $\cdot^{\mathcal{I}}$ is a function that assigns to every $A \in N_C$ a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, to every $r \in N_R$ a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and to every $a \in N_I$ an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$.*

This function is extended to concept descriptions as follows: $(\neg C)^{\mathcal{I}} := \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$; $(C \sqcap D)^{\mathcal{I}} := C^{\mathcal{I}} \cap D^{\mathcal{I}}$; and $(\exists r.C)^{\mathcal{I}} := \{d \in \Delta^{\mathcal{I}} \mid \exists e \in \Delta^{\mathcal{I}} : (d, e) \in r^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\}$. The interpretation $\mathcal{I}$ is a model *of the GCI $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, of the assertion $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$, and of $r(a, b)$ if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$. We write $\mathcal{I} \models \alpha$ if $\mathcal{I}$ is a model of the axiom α, $\mathcal{I} \models \mathcal{T}$ if $\mathcal{I}$ is a model of all GCIs in the TBox $\mathcal{T}$, and $\mathcal{I} \models \mathcal{A}$ if $\mathcal{I}$ is a model of all assertions in the ABox $\mathcal{A}$.*

We assume that all interpretations $\mathcal{I}$ satisfy the *unique name assumption (UNA)*, i.e., for all $a, b \in N_{\mathrm{I}}$ with $a \neq b$ we have $a^{\mathcal{I}} \neq b^{\mathcal{I}}$. We now introduce a temporal query language that generalizes a subset of first-order queries called conjunctive queries [1,8] and the temporal DL $\mathcal{ALC}$-LTL [4]. In the following, we assume (as in [4]) that a subset of the concept and role names is designated as being rigid. The intuition is that the interpretation of the rigid names is not allowed to change over time. Let N_{RC} denote the *rigid concept names*, and N_{RR} the *rigid role names* with $N_{\mathrm{RC}} \subseteq N_{\mathrm{C}}$ and $N_{\mathrm{RR}} \subseteq N_{\mathrm{R}}$. We sometimes call the names in $N_{\mathrm{C}} \setminus N_{\mathrm{RC}}$ and $N_{\mathrm{R}} \setminus N_{\mathrm{RR}}$ *flexible*. As usual, all individual names are implicitly assumed to be rigid.

Definition 2.3. *A* temporal knowledge base *(TKB) $\mathcal{K} = \langle (\mathcal{A}_i)_{0 \leq i \leq n}, \mathcal{T} \rangle$ consists of a finite sequence of ABoxes $\mathcal{A}_i$ and a global TBox $\mathcal{T}$.*

*Let $\mathfrak{I} = (\mathcal{I}_i)_{i \geq 0}$ be an infinite sequence of interpretations $\mathcal{I}_i = (\Delta, \cdot^{\mathcal{I}_i})$ over a fixed non-empty domain Δ (*constant domain assumption*). Then $\mathfrak{I}$ is a* model *of $\mathcal{K}$ (written $\mathfrak{I} \models \mathcal{K}$) if (i) $\mathcal{I}_i \models \mathcal{A}_i$ for all $i, 0 \leq i \leq n$, (ii) $\mathcal{I}_i \models \mathcal{T}$ for all $i \geq 0$, and (iii) $\mathfrak{I}$ respects rigid names, i.e., $x^{\mathcal{I}_i} = x^{\mathcal{I}_j}$ for all $x \in N_I \cup N_{RC} \cup N_{RR}$ and all $i, j \geq 0$.*

We denote by $\mathsf{Ind}(\mathcal{K})$ the set of all individual names occurring in the TKB $\mathcal{K}$. As query language, we use a temporal extension of conjunctive queries.

Definition 2.4. *Let N_V be a set of variables. A* conjunctive query (CQ) *is of the form $\phi = \exists y_1, \ldots, y_m.\psi$, where $y_1, \ldots, y_m \in N_V$ and ψ is a finite conjunction*

of atoms *of the form* $A(z)$ *for* $A \in N_C$ *and* $z \in N_V \cup N_I$ (concept atom)*; or* $r(z_1, z_2)$ *for* $r \in N_R$ *and* $z_1, z_2 \in N_V \cup N_I$ (role atom)*. The empty conjunction is denoted by* true*.* Temporal conjunctive queries (TCQs) *are built from CQs using the constructors* $\neg\phi_1$ *(negation),* $\phi_1 \wedge \phi_2$ *(conjunction),* $\bigcirc\phi_1$ *(next),* $\bigcirc^-\phi_1$ *(previous),* $\phi_1 \mathsf{U} \phi_2$ *(until), and* $\phi_1 \mathsf{S} \phi_2$ *(since).*

We denote the set of individuals occurring in a TCQ ϕ by $\mathsf{Ind}(\phi)$, the set of variables occurring in ϕ by $\mathsf{Var}(\phi)$, and the set of free variables occurring in ϕ by $\mathsf{FVar}(\phi)$. We call a TCQ ϕ with $\mathsf{FVar}(\phi) = \emptyset$ a *Boolean TCQ*. As usual, we use the following abbreviations: $\phi_1 \vee \phi_2$ (disjunction) for $\neg(\neg\phi_1 \wedge \neg\phi_2)$, $\Diamond\phi$ (eventually) for $\mathsf{true}\mathsf{U}\phi$, $\Box\phi$ (always) for $\neg\Diamond\neg\phi$, and analogously for the past: $\Diamond^-\phi$ for $\mathsf{true}\mathsf{S}\phi$, and $\Box^-\phi$ for $\neg\Diamond^-\neg\phi$. A *union of CQs* is a disjunction of CQs.

For our purposes, it is sufficient to define the semantics of CQs and TCQs only for Boolean queries. As usual, it is given using the notion of homomorphisms [8].

Definition 2.5. *Let* $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ *be an interpretation and* ψ *be a Boolean CQ. A mapping* $\pi\colon \mathsf{Var}(\psi) \cup \mathsf{Ind}(\psi) \to \Delta$ *is a* homomorphism *of* ψ *into* $\mathcal{I}$ *if*

- $\pi(a) = a^{\mathcal{I}}$ *for all* $a \in \mathsf{Ind}(\psi)$*;*
- $\pi(z) \in A^{\mathcal{I}}$ *for all concept atoms* $A(z)$ *in* ψ*; and*
- $(\pi(z_1), \pi(z_2)) \in r^{\mathcal{I}}$ *for all role atoms* $r(z_1, z_2)$ *in* ψ*.*

We say that $\mathcal{I}$ *is a* model *of* ψ *(written* $\mathcal{I} \models \psi$*) if there is such a homomorphism. Let now* ϕ *be a Boolean TCQ. For an infinite sequence of interpretations* $\mathfrak{I} = (\mathcal{I}_i)_{i \geq 0}$ *and* $i \geq 0$*, we define* $\mathfrak{I}, i \models \phi$ *by induction on the structure of* ϕ*:*

$$
\begin{array}{ll}
\mathfrak{I}, i \models \exists y_1, \ldots, y_m.\psi & \text{iff } \mathcal{I}_i \models \exists y_1, \ldots, y_m.\psi \\
\mathfrak{I}, i \models \neg\phi_1 & \text{iff } \mathfrak{I}, i \not\models \phi_1 \\
\mathfrak{I}, i \models \phi_1 \wedge \phi_2 & \text{iff } \mathfrak{I}, i \models \phi_1 \text{ and } \mathfrak{I}, i \models \phi_2 \\
\mathfrak{I}, i \models \bigcirc\phi_1 & \text{iff } \mathfrak{I}, i+1 \models \phi_1 \\
\mathfrak{I}, i \models \bigcirc^-\phi_1 & \text{iff } i > 0 \text{ and } \mathfrak{I}, i-1 \models \phi_1 \\
\mathfrak{I}, i \models \phi_1 \mathsf{U} \phi_2 & \text{iff there is some } k \geq i \text{ such that } \mathfrak{I}, k \models \phi_2 \\
 & \quad \text{and } \mathfrak{I}, j \models \phi_1 \text{ for all } j,\ i \leq j < k \\
\mathfrak{I}, i \models \phi_1 \mathsf{S} \phi_2 & \text{iff there is some } k,\ 0 \leq k \leq i \text{ such that } \mathfrak{I}, k \models \phi_2 \\
 & \quad \text{and } \mathfrak{I}, j \models \phi_1 \text{ for all } j,\ k < j \leq i
\end{array}
$$

Given a TKB $\mathcal{K} = \langle (\mathcal{A}_i)_{0 \leq i \leq n}, \mathcal{T} \rangle$*, we say that* $\mathfrak{I}$ *is a* model *of* ϕ *w.r.t.* $\mathcal{K}$ *if* $\mathfrak{I} \models \mathcal{K}$ *and* $\mathfrak{I}, n \models \phi$*. We call* ϕ *satisfiable w.r.t.* $\mathcal{K}$ *if it has a model w.r.t.* $\mathcal{K}$*.*

It should be noted that Boolean TCQs generalize $\mathcal{ALC}$-LTL formulae as introduced in [4]. More precisely, every TCQ that contains only assertions instead of general CQs and contains no past operators ($\bigcirc^-$ or S) is an $\mathcal{ALC}$-LTL formula. $\mathcal{ALC}$-LTL formulae may additionally contain local GCIs $C \sqsubseteq D$. Such a GCI can, however, be expressed by the TCQ $\neg\exists x.A(x)$ if we add the (global) GCIs $A \sqsubseteq C \sqcap \neg D$, $C \sqcap \neg D \sqsubseteq A$ to the TBox. Thus, TCQs together with a global TBox can express all $\mathcal{ALC}$-LTL formulae. TCQs are more expressive than $\mathcal{ALC}$-LTL formulae since CQs like $\exists y.r(y, y)$, which says that there is a loop in the model

without naming the individual which has the loop, can clearly not be expressed in $\mathcal{ALC}$.

Before defining the main inference problem for TCQs to be investigated in this paper, we introduce some notation that will be used later on. The *propositional abstraction* $\widehat{\phi}$ of a TCQ ϕ is built by replacing each CQ occurring in ϕ by a propositional variable such that there is a 1–1 relationship between the CQs $\alpha_1, \dots, \alpha_m$ occurring in ϕ and the propositional variables $p_1, \dots, p_m$ occurring in $\widehat{\phi}$. The formula $\widehat{\phi}$ obtained this way is a propositional LTL-formula [16]. Recall that the semantics of propositional LTL is defined using the notion of an *LTL-structure*, which is an infinite sequence $\mathfrak{J} = (w_i)_{i \geq 0}$ of *worlds* $w_i \subseteq \{p_1, \dots, p_m\}$. The propositional variable p_j is *satisfied* by $\mathfrak{J}$ at time point $i \geq 0$ (written $\mathfrak{J}, i \models p_j$) iff $p_j \in w_i$. The satisfaction of a complex propositional LTL-formula by an LTL-structure is defined as in Definition 2.5.

A *CQ-literal* is a Boolean CQ ψ or a negated Boolean CQ $\neg\psi$. We will often deal with conjunctions ϕ of CQ-literals. Since such a formula ϕ contains no temporal operators, the satisfaction of ϕ by an infinite sequence of interpretations $\mathfrak{I} = (\mathcal{I}_i)_{i \geq 0}$ at time point i only depends on the interpretation $\mathcal{I}_i$. For simplicity, we then often write $\mathcal{I}_i \models \phi$ instead of $\mathfrak{I}, i \models \phi$. By the same argument, we use this notation also for unions of CQs. In this context, it is sufficient to deal with *classical* knowledge bases $\mathcal{K} = \langle \mathcal{A}, \mathcal{T} \rangle$, i.e., temporal knowledge bases with only one ABox, and we similarly write $\mathcal{I}_0 \models \mathcal{K}$ instead of $\mathfrak{I}, 0 \models \mathcal{K}$.

3 The Entailment Problem

We are now ready to introduce the central reasoning problems of this paper, i.e., the problem of finding so-called certain answers to TCQs and the corresponding decision problems.

Definition 3.1. *Let ϕ be a TCQ and $\mathcal{K} = \langle (\mathcal{A}_i)_{0 \leq i \leq n}, \mathcal{T} \rangle$ a temporal knowledge base. The mapping $\mathfrak{a} \colon \mathsf{FVar}(\phi) \to \mathsf{Ind}(\mathcal{K})$ is a* certain answer *to ϕ w.r.t. $\mathcal{K}$ if for every $\mathfrak{I} \models \mathcal{K}$, we have $\mathfrak{I}, n \models \mathfrak{a}(\phi)$, where $\mathfrak{a}(\phi)$ denotes the Boolean TCQ that is obtained from ϕ by replacing the free variables according to $\mathfrak{a}$. The corresponding decision problem is the* recognition problem*, i.e., given $\mathfrak{a}$, ϕ, and $\mathcal{K}$, to check whether $\mathfrak{a}$ is a certain answer to ϕ w.r.t. $\mathcal{K}$. The* (query) entailment problem *is to decide for a Boolean TCQ ϕ and a temporal knowledge base $\mathcal{K} = \langle (\mathcal{A}_i)_{0 \leq i \leq n}, \mathcal{T} \rangle$ whether every model $\mathfrak{I}$ of $\mathcal{K}$ satisfies $\mathfrak{I}, n \models \phi$ (written $\mathcal{K} \models \phi$).*

Note that, for a TCQ ϕ, a temporal knowledge base $\mathcal{K}$, and $i \geq 0$, one can compute all certain answers by enumerating all mappings $\mathfrak{a} \colon \mathsf{FVar}(\phi) \to \mathsf{Ind}(\mathcal{K})$ and then solving the recognition problem for each $\mathfrak{a}$. Since there are $|\mathsf{Ind}(\mathcal{K})|^{|\mathsf{FVar}(\phi)|}$ such mappings, in order to compute the set of certain answers, we have to solve the recognition problem exponentially often.

As described in the introduction, in a situation awareness tool we want to solve the recognition problem for temporal knowledge bases $\mathcal{K} = \langle (\mathcal{A}_i)_{0 \leq i \leq n}, \mathcal{T} \rangle$ and TCQs. The intuition is that the ABoxes $\mathcal{A}_i$ describe our observations about the system's states at time points $i = 0, \dots, n$, where n is the current time point,

and the TCQ describes the situation we want to recognize at time point n for a given instantiation of the free variables in the query (e.g., a certain patient).

Obviously, the entailment problem is a special case of the recognition problem, where $\mathfrak{a}$ is the empty mapping. Conversely, the recognition problem for $\mathfrak{a}$, ϕ, and $\mathcal{K}$ is the same as the entailment problem for $\mathfrak{a}(\phi)$ and $\mathcal{K}$. Thus, these two problems have the same complexity.

Therefore, it is sufficient to analyze the complexity of the *entailment problem*. We consider two kinds of complexity measures: combined complexity and data complexity. For the *combined complexity*, all parts of the input, i.e., the TCQ ϕ and the temporal knowledge base $\mathcal{K}$, are taken into account. For the *data complexity*, the TCQ ϕ and the TBox $\mathcal{T}$ are assumed to be constant, and the complexity is measured only w.r.t. the data, i.e., the sequence of ABoxes. As usual when investigating the data complexity of OBDA [5], we assume that the ABoxes occurring in a temporal knowledge base and the query contain only concept and role names that also occur in the global TBox.

It turns out that it is actually easier to analyze the complexity of the complement of this problem, i.e., *non-entailment* $\mathcal{K} \not\models \phi$. This problem has the same complexity as the *satisfiability problem*. In fact, $\mathcal{K} \not\models \phi$ iff $\neg\phi$ has a model w.r.t. $\mathcal{K}$, and conversely ϕ has a model w.r.t. $\mathcal{K}$ iff $\mathcal{K} \not\models \neg\phi$.

We first analyze the (atemporal) special case of the satisfiability problem where ϕ is a conjunction of CQ-literals. The following result will turn out to be useful also for analyzing the general case.

Theorem 3.2. *Let $\mathcal{K} = \langle \mathcal{A}, \mathcal{T} \rangle$ be a knowledge base and ϕ be a conjunction of CQ-literals. Then deciding whether ϕ has a model w.r.t. $\mathcal{K}$ is* ExpTime*-complete w.r.t. combined complexity and* NP*-complete w.r.t. data complexity.*

Proof (Sketch). The lower bounds easily follow from the known lower bounds for concept satisfiability in $\mathcal{ALC}$ w.r.t. TBoxes [19] and for the data complexity of query answering of Boolean CQs in $\mathcal{ALC}$ [6]. To check whether there is an interpretation $\mathcal{I}$ with $\mathcal{I} \models \mathcal{K}$ and $\mathcal{I} \models \phi$, we reduce this problem to a query non-entailment problem of known complexity. First, we instantiate the non-negated CQs in ϕ by omitting the existential quantifiers and replacing the variables by fresh individual names. The set $\mathcal{A}'$ of the resulting atoms can thus be viewed as an additional ABox that restricts the interpretation $\mathcal{I}$. The above problem is thus equivalent to finding an interpretation $\mathcal{I}$ with $\mathcal{I} \models \langle \mathcal{A} \cup \mathcal{A}', \mathcal{T} \rangle$ and $\mathcal{I} \not\models \rho$, where ρ is the union of Boolean CQs that results from negating the conjunction of all negated CQs in ϕ. This is the same as asking whether the knowledge base $\langle \mathcal{A} \cup \mathcal{A}', \mathcal{T} \rangle$ does not entail the union of conjunctive queries ρ. The complexity of this kind of entailment problems is known: it is ExpTime-complete w.r.t. combined complexity [7,13] and co-NP-complete w.r.t. data complexity [15]. □

We now describe an approach to solving the satisfiability problem (and thus the non-entailment problem) in general. The basic idea is to reduce the problem to two separate satisfiability problems, similar to what was done for $\mathcal{ALC}$-LTL in Lemma 4.3 of [4]. Let $\mathcal{K} = \langle (\mathcal{A}_i)_{0 \le i \le n}, \mathcal{T} \rangle$ be a TKB and ϕ be a Boolean TCQ, for which we want to decide whether ϕ has a model w.r.t. $\mathcal{K}$. Recall

that the propositional abstraction $\widehat{\phi}$ of ϕ contains the propositional variables $p_1, \ldots, p_m$ in place of the CQs $\alpha_1, \ldots, \alpha_m$ occurring in ϕ. We assume in the following that α_i was replaced by p_i for all i, $1 \leq i \leq m$. We now consider a set $\mathcal{S} \subseteq 2^{\{p_1,\ldots,p_m\}}$, which intuitively specifies the worlds that are allowed to occur in an LTL-structure satisfying $\widehat{\phi}$. To express this restriction, we define the propositional LTL-formula

$$\widehat{\phi}_{\mathcal{S}} := \widehat{\phi} \wedge \Box^{-}\Box \left(\bigvee_{X \in \mathcal{S}} \left(\bigwedge_{p \in X} p \wedge \bigwedge_{p \notin X} \neg p \right) \right).^{2}$$

If ϕ has a model w.r.t. $\mathcal{K}$, i.e., there is a sequence of interpretations $\mathfrak{I} = (\mathcal{I}_i)_{i \geq 0}$ that respects rigid names, is a model of $\mathcal{K}$, and satisfies $\mathfrak{I}, n \models \phi$, then there exist a set $\mathcal{S} \subseteq 2^{\{p_1,\ldots,p_m\}}$ and a propositional LTL-structure that satisfies $\widehat{\phi}_{\mathcal{S}}$ at time point n. In fact, for each interpretation $\mathcal{I}_i$ of $\mathfrak{I}$, we set $X_i := \{p_j \mid 1 \leq j \leq m \text{ and } \mathcal{I}_i \text{ satisfies } \alpha_j\}$, and then take $\mathcal{S} := \{X_i \mid i \geq 0\}$. We say that $\mathcal{S}$ is *induced* by $\mathfrak{I}$. The fact that $\mathfrak{I}$ satisfies ϕ at time point n implies that its propositional abstraction satisfies $\widehat{\phi}_{\mathcal{S}}$ at time point n, where the *propositional abstraction* $\widehat{\mathfrak{I}} = (w_i)_{i \geq 0}$ of $\mathfrak{I}$ is defined by $w_i := X_i$ for all $i \geq 0$. However, guessing a set $\mathcal{S}$ and then testing whether the induced LTL-formula $\widehat{\phi}_{\mathcal{S}}$ is satisfiable at time point n is not sufficient for checking whether ϕ has a model w.r.t. $\mathcal{K}$. We must also check whether the guessed set $\mathcal{S}$ can indeed be induced by some sequence of interpretations that is a model of $\mathcal{K}$. The following definition introduces a condition that needs to be satisfied for this to hold.

Definition 3.3. *Given a set $\mathcal{S} = \{X_1, \ldots, X_k\} \subseteq 2^{\{p_1,\ldots,p_m\}}$ and a mapping $\iota\colon \{0, \ldots, n\} \to \{1, \ldots, k\}$, we say that $\mathcal{S}$ is* r-consistent *w.r.t. ι and $\mathcal{K}$ if there exist interpretations $\mathcal{J}_1, \ldots, \mathcal{J}_k$, $\mathcal{I}_0, \ldots, \mathcal{I}_n$ such that*

- *the interpretations share the same domain and respect rigid names;*[3]
- *the interpretations are models of $\mathcal{T}$;*
- *for i, $0 \leq i \leq k$, $\mathcal{J}_i$ is a model of $\chi_i := \bigwedge_{p_j \in X_i} \alpha_j \wedge \bigwedge_{p_j \notin X_i} \neg \alpha_j$; and*
- *for i, $0 \leq i \leq n$, $\mathcal{I}_i$ is a model of $\mathcal{A}_i$ and $\chi_{\iota(i)}$.*

The intuition underlying this definition is the following. The existence of the interpretation $\mathcal{J}_i$ ($1 \leq i \leq k$) ensures that the conjunction χ_i of the CQ-literals specified by X_i is consistent. In fact, a set $\mathcal{S}$ containing a set X_i for which this does not hold cannot be induced by a sequence of interpretations. The interpretations $\mathcal{I}_i$ ($0 \leq i \leq n$) are supposed to constitute the first $n + 1$ interpretations in such a sequence. In addition to inducing a set $X_{\iota(i)} \in \mathcal{S}$ and thus satisfying the corresponding conjunction $\chi_{\iota(i)}$, the interpretation $\mathcal{I}_i$ must thus also satisfy the ABox $\mathcal{A}_i$. The first and the second condition ensure that a sequence of interpretations built from $\mathcal{J}_1, \ldots, \mathcal{J}_k, \mathcal{I}_0, \ldots, \mathcal{I}_n$ respects rigid names and satisfies

[2] Note that a formula $\Box^{-}\Box\psi$ is satisfied iff ψ holds at all time points.

[3] This is defined analogously to the case of sequences of interpretations (Definition 2.3).

the global TBox $\mathcal{T}$. Note that we can use Theorem 3.2 to check whether interpretations satisfying the last three conditions of Definition 3.3 exist. As we will see below, the difficulty lies in ensuring that they also satisfy the first condition.

Satisfaction of the temporal structure of ϕ by a sequence of interpretations built this way is ensured by testing $\widehat{\phi}_{\mathcal{S}}$ for satisfiability, which can basically be done using algorithms for testing satisfiability in propositional LTL [23].

Lemma 3.4. *The TCQ ϕ has a model w.r.t. the TKB $\mathcal{K}$ iff there is a set $\mathcal{S} = \{X_1, \ldots, X_k\} \subseteq 2^{\{p_1,\ldots,p_m\}}$ and a mapping $\iota\colon \{0,\ldots,n\} \to \{1,\ldots,k\}$ such that*

1. *$\mathcal{S}$ is r-consistent w.r.t. ι and $\mathcal{K}$, and*
2. *there is an LTL-structure $\mathfrak{J} = (w_i)_{i\geq 0}$ such that $\mathfrak{J}, n \models \widehat{\phi}_{\mathcal{S}}$ and $w_i = X_{\iota(i)}$ for all i, $0 \leq i \leq n$.*

The proof of this lemma is similar to, but more involved than the proof of a similar characterization for satisfiability in $\mathcal{ALC}$-LTL [4].

As shown later, the overall complexity of the satisfiability problem depends on which symbols are allowed to be rigid. To achieve these complexity results, we obtain the set $\mathcal{S}$ and the function ι either by enumeration, guessing, or direct construction, depending on the case under consideration. Given $\mathcal{S}$ and ι, it remains to check the two conditions of the lemma. To check the second condition, we construct a Büchi automaton similar to the standard construction for satisfiability of LTL-formulae [23]. Emptiness of this automaton is equivalent to satisfiability of $\widehat{\phi}_{\mathcal{S}}$. The details can be found in [3].

The main difference to the standard construction is the additional condition $w_i = X_{\iota(i)}$ for i, $0 \leq i \leq n$. We check this by attaching a counter taking values from $\{0,\ldots,n+1\}$ to the states of the automaton. Transitions where the counter is $i < n+1$ check if the current world corresponds to $X_{\iota(i)}$ and increase the counter by 1. At $i = n$, we ensure that $\widehat{\phi}_{\mathcal{S}}$ is satisfied. Similar to what is done in [4], we do not construct the automaton directly for $\widehat{\phi}_{\mathcal{S}}$, which would yield an automaton of double-exponential size in the size of ϕ, but rather for $\widehat{\phi}$. The additional restrictions of $\widehat{\phi}_{\mathcal{S}}$ are enforced by restricting this automaton to states that satisfy a world from $\mathcal{S}$. The size of the constructed automaton only depends linearly on the number n of input ABoxes, which is important for the results about data complexity, and exponentially on the size of ϕ. Furthermore, emptiness of Büchi automata can be checked in polynomial time in the size of the automaton [23].

Lemma 3.5. *Given a set $\mathcal{S} = \{X_1, \ldots, X_k\} \subseteq 2^{\{p_1,\ldots,p_m\}}$ and a mapping $\iota\colon \{0,\ldots,n\} \to \{1,\ldots,k\}$, the problem of deciding the existence of an LTL-structure $\mathfrak{J} = (w_i)_{i\geq 0}$ such that $\mathfrak{J}, n \models \widehat{\phi}_{\mathcal{S}}$ and $w_i = X_{\iota(i)}$ for all i, $0 \leq i \leq n$, is in* ExpTime *w.r.t. combined complexity and in* P *w.r.t. data complexity.*

For the r-consistency test, we need to use different constructions depending on which symbols are allowed to be rigid. Using these constructions, we obtain the complexity results for the entailment problem shown in Table 1. Note that rigid concept names can be simulated by rigid role names [4], which is why there are

Table 1. The complexity of the entailment problem for TCQs

	Data complexity	*Combined complexity*
$N_{\mathrm{RC}} = N_{\mathrm{RR}} = \emptyset$	co-NP-complete	ExpTime-complete
$N_{\mathrm{RC}} \neq \emptyset$, $N_{\mathrm{RR}} = \emptyset$	co-NP-complete	co-NExpTime-complete
$N_{\mathrm{RR}} \neq \emptyset$	co-NP-hard/in ExpTime	2-ExpTime-complete

only three cases to consider. The lower bounds can be obtained by simple reductions from the atemporal entailment problem [6] and the satisfiability problem of $\mathcal{ALC}$-LTL [4]. In the following sections, we only present the ideas for the upper bounds in the most interesting case (no rigid role names, but rigid concept names). For the other two cases, the proofs are quite similar to the ones for $\mathcal{ALC}$-LTL [4]. For rigid concepts, the proofs still follow the lines of the proofs in [4], but need considerably more effort to deal with CQs instead of assertions (see [3] for more details).

4 Data Complexity for the Case of Rigid Concepts

To obtain an upper bound for the data complexity of the non-entailment problem in the case where $N_{\mathrm{RC}} \neq \emptyset$ and $N_{\mathrm{RR}} = \emptyset$, we consider the conditions of Lemma 3.4 in more detail. First, note that, since $\mathcal{S} \subseteq 2^{\{p_1,\dots,p_m\}}$ is of constant size w.r.t. the input ABoxes and $\iota\colon \{0,\dots,n\} \to \{1,\dots,k\}$ is of size linear in n (the number of ABoxes), guessing $\mathcal{S}$ and ι can be done in NP. Additionally, according to Lemma 3.5, LTL-satisfiability can be tested in P.

We now show that the r-consistency of $\mathcal{S}$ w.r.t. ι and $\mathcal{K}$ can be checked in NP, which yields the desired data complexity of co-NP for the entailment problem. We use a renaming technique similar to the one employed in [4]. For every i, $1 \leq i \leq k$, and every *flexible* concept name A (every role name r) occurring in ϕ or in $\mathcal{T}$, we introduce a copy $A^{(i)}$ ($r^{(i)}$), which is a fresh concept (role) name. We call $A^{(i)}$ ($r^{(i)}$) the i-th copy of A (r). The CQ $\alpha^{(i)}$ (the GCI $\beta^{(i)}$) is obtained from a CQ α (a GCI β) by replacing every occurrence of a flexible name by its i-th copy. Similarly, for $1 \leq \ell \leq k$, the conjunction $\chi_\ell^{(i)}$ is obtained from χ_ℓ (see Definition 3.3) by replacing each CQ α_j by $\alpha_j^{(i)}$.

The basic idea is to decide the existence of models of the conjunctions of CQ-literals $\gamma_i \wedge \chi_{\mathcal{S}}$ w.r.t. the TBox $\mathcal{T}_{\mathcal{S}}$, where

$$\gamma_i := \bigwedge_{\alpha \in \mathcal{A}_i} \alpha^{(\iota(i))}, \quad \chi_{\mathcal{S}} := \bigwedge_{1 \leq i \leq k} \chi_i^{(i)}, \quad \mathcal{T}_{\mathcal{S}} := \{\beta^{(i)} \mid \beta \in \mathcal{T} \text{ and } 1 \leq i \leq k\}.$$

One can see from the proof of Theorem 3.2 that this problem can be decided in NP in the size of the input ABoxes. The main reason is that the negated CQs do not depend on the input ABoxes. In fact, negated CQs only occur in $\chi_{\mathcal{S}}$, which only depends on the query ϕ. Thus, the union of CQs ρ constructed in the proof of Theorem 3.2 does not depend on the input ABoxes and the same is true for the TBox $\mathcal{T}_{\mathcal{S}}$.

However, for r-consistency we have to make sure that rigid consequences of the form $A(a)$ for a rigid concept name A and an individual name a are shared between these conjunctions of CQ-literals. Let $\mathsf{RCon}(\mathcal{T})$ denote the rigid concept names occurring in $\mathcal{T}$. Similar to what was done in Lemma 6.3 of [4], we now guess a set $\mathcal{D} \subseteq 2^{\mathsf{RCon}(\mathcal{T})}$ and a mapping $\tau\colon \mathsf{Ind}(\phi) \cup \mathsf{Ind}(\mathcal{K}) \to \mathcal{D}$. The idea is that $\mathcal{D}$ fixes the combinations of rigid concept names that occur in the models of $\gamma_i \wedge \chi_{\mathcal{S}}$ and τ assigns to each individual name one such combination. Note that $\mathcal{D}$ only depends on $\mathcal{T}$ and τ is of size linear in the size of the input ABoxes, which is why we can guess $\mathcal{D}$ and τ in NP w.r.t. data complexity. We now define

$$\chi_\tau := \bigwedge_{a \in \mathsf{Ind}(\phi) \cup \mathsf{Ind}(\mathcal{K})} \left(\bigwedge_{A \in \tau(a)} A(a) \quad \wedge \quad \bigwedge_{A \in \mathsf{RCon}(\mathcal{T}) \setminus \tau(a)} A'(a) \right),$$

where A' is a rigid concept name that is equivalent to $\neg A$ in $\mathcal{T}$.[4] Note that χ_τ is of polynomial size w.r.t. the size of the input ABoxes.

We need one more notation to formulate the main lemma of this section. We say that an interpretation $\mathcal{I}$ *respects* $\mathcal{D}$ if

$$\mathcal{D} = \{Y \subseteq \mathsf{RCon}(\mathcal{T}) \mid \text{there is a } d \in \Delta^{\mathcal{I}} \text{ such that } d \in (C_Y)^{\mathcal{I}}\},$$

where $C_Y := \bigsqcap_{A \in Y} A \ \sqcap \ \bigsqcap_{A \in \mathsf{RCon}(\mathcal{T}) \setminus Y} \neg A$.

Lemma 4.1. *If $N_{RC} \neq \emptyset$ and $N_{RR} = \emptyset$, then $\mathcal{S}$ is r-consistent w.r.t. ι and $\mathcal{K}$ iff there exist $\mathcal{D} \subseteq 2^{\mathsf{RCon}(\mathcal{T})}$ and $\tau\colon \mathsf{Ind}(\phi) \cup \mathsf{Ind}(\mathcal{K}) \to \mathcal{D}$ such that each of the conjunctions $\gamma_i \wedge \chi_{\mathcal{S}} \wedge \chi_\tau$, $0 \leq i \leq n$, has a model w.r.t. $\mathcal{T}_{\mathcal{S}}$ that respects $\mathcal{D}$.*

Proof (Sketch). For the "if" direction, assume that $\mathcal{I}_i$ are the required models for $\gamma_i \wedge \chi_{\mathcal{S}} \wedge \chi_\tau$. Similar to the proof of Lemma 6.3 in [4], we can assume w.l.o.g. that their domains Δ_i are countably infinite and for each $Y \in \mathcal{D}$ there are countably infinitely many elements $d \in (C_Y)^{\mathcal{I}_i}$. This is a consequence of the Löwenheim-Skolem theorem and the fact that the countably infinite disjoint union of $\mathcal{I}_i$ with itself is again a model of $\gamma_i \wedge \chi_{\mathcal{S}} \wedge \chi_\tau$. The latter follows from the observation that for any CQ there is a homomorphism into $\mathcal{I}_i$ iff there is a homomorphism into the disjoint union of $\mathcal{I}_i$ with itself.

Consequently, we can partition the domains Δ_i into the countably infinite sets $\Delta_i(Y) := \{d \in \Delta_i \mid d \in (C_Y)^{\mathcal{I}_i}\}$ for $Y \in \mathcal{D}$. It is now easy to see that the domains Δ_i are essentially the same up to isomorphisms between Δ_i and Δ_j for $0 \leq i, j \leq n$ that relate the elements of $\Delta_i(Y)$ to those of $\Delta_j(Y)$, and respect the individual names, i.e., map each $a^{\mathcal{I}_i}$ to $a^{\mathcal{I}_j}$. We can now construct the models required by Definition 3.3 from the models $\mathcal{I}_i$ by appropriately relating the flexible names and their copies. For example, interpreting the rigid concept names as in $\mathcal{I}_i$ and the flexible names as their $\iota(i)$-th copies in $\mathcal{I}_i$ yields a model

[4] We can assume w.l.o.g. that for each rigid concept name in $\mathcal{T}$, there is a rigid concept name equivalent to its negation in $\mathcal{T}$. We can introduce them if needed while multiplying the size of the TBox by at most 2. We cannot include $\neg A(a)$ in χ_τ since this could result in polynomially many negated CQs in the size of the ABoxes.

of $\chi_{\iota(i)}$ w.r.t. $\langle \mathcal{A}_i, \mathcal{T} \rangle$, and similarly for the models of χ_j and $\mathcal{T}$ for $1 \leq j \leq k$. These models share the same domain and respect rigid names. Note that the interpretation of the names in $N_{\mathrm{RC}} \setminus \mathsf{RCon}(\mathcal{T})$ and $N_{\mathrm{I}} \setminus (\mathsf{Ind}(\phi) \cup \mathsf{Ind}(\mathcal{K}))$ is irrelevant and can be fixed arbitrarily.

For the "only if" direction, it is easy to see that one can combine the interpretations $\mathcal{I}_i$, $\mathcal{J}_1$, ..., $\mathcal{J}_k$ from Definition 3.3 to a model $\mathcal{I}'_i$ of $\gamma_i \wedge \chi_{\mathcal{S}}$ w.r.t. $\mathcal{T}_{\mathcal{S}}$ by interpreting the j-th copy of a flexible name as the original name in $\mathcal{J}_j$. For $a \in \mathsf{Ind}(\phi) \cup \mathsf{Ind}(\mathcal{K})$, we define $\tau(a) := Y \subseteq \mathsf{RCon}(\mathcal{T})$ iff $a \in (C_Y)^{\mathcal{I}_0}$. Furthermore, we let $\mathcal{D}$ contain all those sets $Y \subseteq \mathsf{RCon}(\mathcal{T})$ such that there is a $d \in (C_Y)^{\mathcal{I}'_i}$ for some $0 \leq i \leq n$. To obtain models of $\gamma_i \wedge \chi_{\mathcal{S}} \wedge \chi_\tau$ w.r.t. $\mathcal{T}_{\mathcal{S}}$ that respect $\mathcal{D}$, we still need to ensure that all $Y \in \mathcal{D}$ are represented in each of the models $\mathcal{I}'_i$. To do this, we construct the disjoint union $\mathcal{I}''_i$ of $\mathcal{I}'_i$ with all other $\mathcal{I}'_j$ for $0 \leq j \leq n$. It remains to show that this interpretation is still a model of $\mathcal{T}_{\mathcal{S}}$ and the conjunction $\gamma_i \wedge \chi_{\mathcal{S}} \wedge \chi_\tau$. This can be seen as follows. For the non-negated CQs in this conjunction, clearly there is a homomorphism into $\mathcal{I}''_i$ if there is one into $\mathcal{I}'_i$. For the negated CQs in $\chi_{\mathcal{S}}$, we need the additional assumption that each of them is *connected* in the sense that the variables and individual names are related by roles (see [18] or [3] for an exact definition). It follows from a result in [21] that this is without loss of generality (see [3]). Given this assumption, the non-existence of a homomorphism into any of the components of $\mathcal{I}''_i$ clearly implies the non-existence of a homomorphism into their disjoint union $\mathcal{I}''_i$. □

It remains to show that we can check the existence of a model of $\gamma_i \wedge \chi_{\mathcal{S}} \wedge \chi_\tau$ w.r.t. $\mathcal{T}_{\mathcal{S}}$ that respects $\mathcal{D}$ in nondeterministic polynomial time. For this, observe that the restriction imposed by $\mathcal{D}$ can equivalently be expressed as

$$\chi_{\mathcal{D}} := (\neg \exists x.A_{\mathcal{D}}(x)) \wedge \bigwedge_{Y \in \mathcal{D}} \exists x.A_Y(x),$$

where A_Y and $A_{\mathcal{D}}$ are fresh concept names that are restricted by adding the GCIs $A_Y \sqsubseteq C_Y$, $C_Y \sqsubseteq A_Y$ for each $Y \in \mathcal{D}$, and $A_{\mathcal{D}} \sqsubseteq \bigsqcap_{Y \in \mathcal{D}} \neg A_Y$, $\bigsqcap_{Y \in \mathcal{D}} \neg A_Y \sqsubseteq A_{\mathcal{D}}$ to $\mathcal{T}_{\mathcal{S}}$. We call the resulting TBox $\mathcal{T}'_{\mathcal{S}}$. Since $\chi_{\mathcal{D}}$ and $\mathcal{T}'_{\mathcal{S}}$ do not depend on the input ABoxes, by Theorem 3.2 we can check the consistency of $\gamma_i \wedge \chi_{\mathcal{S}} \wedge \chi_\tau \wedge \chi_{\mathcal{D}}$ w.r.t. $\mathcal{T}'_{\mathcal{S}}$ in NP w.r.t. data complexity.

Theorem 4.2. *If $N_{RC} \neq \emptyset$ and $N_{RR} = \emptyset$, then the entailment problem is in co-*NP *w.r.t. data complexity.*

5 Combined Complexity for the Case of Rigid Concepts

Unfortunately, the approach used in the previous section does not yield a *combined complexity* of co-NExpTime. The reason is that the conjunctions $\chi_{\mathcal{S}}$ and $\chi_{\mathcal{D}}$ are of exponential size in the size of ϕ, and thus Theorem 3.2 only yields an upper bound of 2-ExpTime. In this section, we describe a different approach with a combined complexity of co-NExpTime.

As a first step, we rewrite the Boolean TCQ ϕ into a Boolean TCQ ψ of linear size in the size of ϕ and $\mathcal{K}$ such that answering ϕ at time point n is equivalent

to answering ψ at time point 0 w.r.t. a trivial sequence of ABoxes. This is done by compiling the ABoxes into the query and postponing the query ϕ using the $\bigcirc$-operator (see [3] for details). We can thus focus on deciding whether a Boolean TCQ ϕ has a model w.r.t. a TKB $\mathcal{K} = \langle \emptyset, \mathcal{T} \rangle$ that has only one empty ABox in the sequence. Note that this compilation approach does not allow us to obtain a low *data complexity* for the entailment problem since after encoding the ABoxes into ϕ the size of $\chi_{\mathcal{S}}$ is exponential in the size of the ABoxes.

We now again analyze how to check the two conditions in Lemma 3.4. First, observe that guessing $\mathcal{S} = \{X_1, \dots, X_k\} \subseteq 2^{\{p_1,\dots,p_m\}}$ can be done in nondeterministic exponential time in the size of ϕ. Furthermore, by Lemma 3.5, the LTL-satisfiability test required by the second condition can be realized in ExpTime. It remains to determine the complexity of testing r-consistency of $\mathcal{S}$ w.r.t. $\mathcal{K} = \langle \emptyset, \mathcal{T} \rangle$. Similarly to the approach used in the previous section and to the proof of Lemma 6.3 in [4], we start by guessing a set $\mathcal{D} \subseteq 2^{\mathsf{RCon}(\mathcal{T})}$ and a mapping $\tau \colon \mathsf{Ind}(\phi) \to \mathcal{D}$. Since $\mathcal{D}$ is of size exponential in $\mathcal{T}$ and τ is of size polynomial in the size of ϕ and $\mathcal{T}$, guessing $\mathcal{D}$ and τ can also be done in NExpTime. By Lemma 4.1, it suffices to test whether $\chi_{\mathcal{S}} \wedge \chi_\tau$ has a model w.r.t. $\mathcal{T}_{\mathcal{S}}$ that respects $\mathcal{D}$. Instead of applying Theorem 3.2 directly to this problem, which would yield a complexity of 2-ExpTime, we split the problem into separate sub-problems for each component χ_i of $\chi_{\mathcal{S}}$. The correctness of this approach is stated in the next lemma. For the special case of $\mathcal{ALC}$-LTL, this was shown in Lemma 6.3 in [4]. The proof for the general case is similar to the proof of Lemma 4.1 above.

Lemma 5.1. *If $N_{RC} \neq \emptyset$ and $N_{RR} = \emptyset$, then $\mathcal{S}$ is r-consistent w.r.t. $\mathcal{K} = \langle \emptyset, \mathcal{T} \rangle$ iff there exist $\mathcal{D} \subseteq 2^{\mathsf{RCon}(\mathcal{T})}$ and $\tau \colon \mathsf{Ind}(\phi) \to \mathcal{D}$ such that each of the conjunctions $\widehat{\chi}_i := \chi_i \wedge \chi_\tau$, $1 \leq i \leq k$, has a model w.r.t. $\mathcal{K}$ that respects $\mathcal{D}$.*

Note that the size of each $\widehat{\chi}_i$ is polynomial in the size of ϕ and $\mathcal{T}$ and the number k of these conjunctions is exponential in the size of ϕ. Thus, it is enough to show that the existence of a model of $\widehat{\chi}_i$ w.r.t. $\mathcal{K}$ that respects $\mathcal{D}$ can be checked in exponential time in the size of ϕ and $\mathcal{T}$. Similar to the proof of Theorem 3.2, we can reduce this problem to a non-entailment problem for a union of Boolean CQs: there is an interpretation that is a model of $\widehat{\chi}_i$ and $\mathcal{T}$ and respects $\mathcal{D}$ iff there is a model of $\langle \mathcal{A}, \mathcal{T} \rangle$ that respects $\mathcal{D}$ and is not a model of ρ (written $\langle \mathcal{A}, \mathcal{T} \rangle \not\models \rho$ w.r.t. $\mathcal{D}$), where $\mathcal{A}$ is an ABox obtained by instantiating the non-negated CQs of $\widehat{\chi}_i$ with fresh individual names and ρ is a union of CQs constructed from the negated CQs of $\widehat{\chi}_i$.

It thus suffices to show that we can decide query non-entailment $\langle \mathcal{A}, \mathcal{T} \rangle \not\models \rho$ w.r.t. $\mathcal{D}$ in time exponential in the size of $\mathcal{A}$, $\mathcal{T}$, and ρ. To this purpose, we further reduce this problem following an idea from [13]. There, the notion of a *spoiler* is introduced. A spoiler is an $\mathcal{ALC}^{\cap}$-knowledge base that states properties that must be satisfied such that a query is not entailed by a knowledge base.[5] It is shown that $\langle \mathcal{A}, \mathcal{T} \rangle \not\models \rho$ iff there is a spoiler $\langle \mathcal{A}', \mathcal{T}' \rangle$ for $\langle \mathcal{A}, \mathcal{T} \rangle$ such that $\langle \mathcal{A} \cup \mathcal{A}', \mathcal{T} \cup \mathcal{T}' \rangle$ is consistent. Additionally, all spoilers can be computed in time exponential in the size of $\langle \mathcal{A}, \mathcal{T} \rangle$ and ρ, and each spoiler is of polynomial size.

[5] $\mathcal{ALC}^{\cap}$ extends $\mathcal{ALC}$ by role conjunctions of the form $r_1 \cap \dots \cap r_n$ for $r_1, \dots, r_n \in N_{\mathrm{R}}$.

We show in [3] that the above reduction is still correct in the presence of $\mathcal{D}$, i.e., we have $\langle \mathcal{A}, \mathcal{T} \rangle \not\models \rho$ w.r.t. $\mathcal{D}$ iff there is a spoiler $\langle \mathcal{A}', \mathcal{T}' \rangle$ for $\langle \mathcal{A}, \mathcal{T} \rangle$ such that there is a model of $\langle \mathcal{A} \cup \mathcal{A}', \mathcal{T} \cup \mathcal{T}' \rangle$ that respects $\mathcal{D}$. It now remains to show that the existence of such a model can be checked in exponential time in the size of $\langle \mathcal{A} \cup \mathcal{A}', \mathcal{T} \cup \mathcal{T}' \rangle$, and therefore in exponential time in the size of ϕ and $\mathcal{T}$.

For classical $\mathcal{ALC}^{\cap}$-knowledge bases, the consistency problem (without $\mathcal{D}$) is ExpTime-complete [22]. The complexity does not increase for checking the existence of a model of a Boolean $\mathcal{ALC}^{\cap}$-knowledge base that respects $\mathcal{D}$.[6] We show this in [3] using a notion of *quasimodels* similar to the one in [4], but extended to deal with role conjunctions. The main difference is that we must introduce additional concept names that function as so-called *pebbles*, which mark elements that have specific role predecessors, an idea borrowed from [9,10,14].

Lemma 5.2. *Let $\mathcal{B}$ be a Boolean $\mathcal{ALC}^{\cap}$-knowledge base of size n, $A_1, \ldots, A_k$ be concept names occurring in $\mathcal{B}$, and $\mathcal{D} \subseteq 2^{\{A_1,\ldots,A_k\}}$. Then the existence of a model of $\mathcal{B}$ that respects $\mathcal{D}$ can be decided in time exponential in n.*

Combining the reductions of this section, we get the desired complexity result.

Theorem 5.3. *If $N_{RC} \neq \emptyset$ and $N_{RR} = \emptyset$, then the entailment problem is in co-*NExpTime *w.r.t. combined complexity.*

6 Conclusions

We have introduced a new temporal query language that extends the temporal DL $\mathcal{ALC}$-LTL to using conjunctive queries as atoms. Our complexity results on the entailment problem for such queries w.r.t. temporal knowledge bases are summarized in Table 1. Without any rigid names, we observed that entailment of TCQs is as hard as entailment of CQs w.r.t. atemporal $\mathcal{ALC}$-knowledge bases, i.e., in this case adding temporal operators to the query language does not increase the complexity. However, if we allow for rigid concept names (but no rigid role names), the picture changes. While the data complexity remains the same as in the atemporal case, the combined complexity of query entailment increases to co-NExpTime, i.e., the non-entailment problem is as hard as satisfiability in $\mathcal{ALC}$-LTL. If we further add rigid role names, the combined complexity (of non-entailment) again increases in accordance with the complexity of satisfiability in $\mathcal{ALC}$-LTL. For data complexity, it is still unclear whether adding rigid role names results in an increase. We have shown an upper bound of ExpTime (which is one exponential better than the combined complexity), but the only lower bound we have is the trivial one of co-NP.

Further work will include trying to close this gap. Moreover, it would be interesting to consider temporal queries based on inexpressive DLs such as DL-Lite [5], and check under what conditions query answering can be realized using classical (temporal or atemporal) database techniques.

[6] Boolean knowledge bases generalize ABoxes and TBoxes by allowing arbitrary Boolean combinations of axioms instead of only conjunctions.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Baader, F., Bauer, A., Baumgartner, P., Cregan, A., Gabaldon, A., Ji, K., Lee, K., Rajaratnam, D., Schwitter, R.: A novel architecture for situation awareness systems. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS, vol. 5607, pp. 77–92. Springer, Heidelberg (2009)
3. Baader, F., Borgwardt, S., Lippmann, M.: On the complexity of temporal query answering. LTCS-Report 13-01, Technische Universität Dresden, Germany (2012), `http://lat.inf.tu-dresden.de/research/reports.html`
4. Baader, F., Ghilardi, S., Lutz, C.: LTL over description logic axioms. ACM Trans. Comput. Log. 13(3) (2012)
5. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., Rosati, R.: Ontologies and databases: The *DL-Lite* approach. In: Tessaris, S., Franconi, E., Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M.-C., Schmidt, R.A. (eds.) Reasoning Web. LNCS, vol. 5689, pp. 255–356. Springer, Heidelberg (2009)
6. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Data complexity of query answering in description logics. In: Proc. KR 2006 (2006)
7. Calvanese, D., De Giacomo, G., Lenzerini, M.: On the decidability of query containment under constraints. In: Proc. PODS 1998 (1998)
8. Chandra, A.K., Merlin, P.M.: Optimal implementation of conjunctive queries in relational data bases. In: Proc. STOC 1977 (1977)
9. Danecki, R.: Nondeterministic propositional dynamic logic with intersection is decidable. In: Skowron, A. (ed.) SCT 1984. LNCS, vol. 208, pp. 34–53. Springer, Heidelberg (1985)
10. De Giacomo, G., Massacci, F.: Combining deduction and model checking into tableaux and algorithms for Converse-PDL. Inform. Comput. 162(1-2) (2000)
11. Decker, S., Erdmann, M., Fensel, D., Studer, R.: Ontobroker: Ontology based access to distributed and semi-structured information. In: Proc. DS 1999 (1999)
12. Endsley, M.R.: Toward a theory of situation awareness in dynamic systems. Human Factors 37(1) (1995)
13. Lutz, C.: The complexity of conjunctive query answering in expressive description logics. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 179–193. Springer, Heidelberg (2008)
14. Massacci, F.: Decision procedures for expressive description logics with intersection, composition, converse of roles and role identity. In: Proc. IJCAI 2001 (2001)
15. Ortiz, M., Calvanese, D., Eiter, T.: Characterizing data complexity for conjunctive query answering in expressive description logics. In: Proc. AAAI 2006 (2006)
16. Pnueli, A.: The temporal logic of programs. In: Proc. FOCS 1977 (1977)
17. Poggi, A., Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Linking data to ontologies. J. Data Sem. X (2008)
18. Rudolph, S., Glimm, B.: Nominals, inverses, counting, and conjunctive queries or: Why infinity is your friend! J. Artif. Intell. Res. 39(1) (2010)
19. Schild, K.: A correspondence theory for terminological logics: Preliminary report. In: Proc. IJCAI 1991 (1991)
20. Schmidt-Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artif. Intell. 48(1) (1991)
21. Tessaris, S.: Questions and Answers: Reasoning and Querying in Description Logic. Ph.D. thesis, University of Manchester (2001)
22. Tobies, S.: Complexity Results and Practical Algorithms for Logics in Knowledge Representation. Ph.D. thesis, RWTH Aachen (2001)
23. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inform. Comput. 155(1) (1994)

Verifying Refutations with Extended Resolution

Marijn J.H. Heule, Warren A. Hunt Jr., and Nathan Wetzler*

The University of Texas at Austin

Abstract. Modern SAT solvers use preprocessing and inprocessing techniques that are not solely based on resolution; existing unsatisfiability proof formats do not support SAT solvers using such techniques. We present a new proof format for checking unsatisfiability proofs produced by SAT solvers that use techniques such as extended resolution and blocked clause addition. Our new format was designed with three goals: proofs should be easy to generate, proofs should be compact, and validating proofs must be simple. We show how existing preprocessors and solvers can be modified to generate proofs in our new format. Additionally, we implemented a mechanically-verified proof checker in ACL2 and a proof checker in C for the proposed format.

1 Introduction

Satisfiability (SAT) solvers have become the core search engine in many tools used for combinational [1,2] and sequential equivalence checking [3,4], bounded [5] and unbounded model checking [6], and debugging [7]; thus, it is crucial that SAT solvers produce correct results. Presently, some of the best solvers use implementation techniques for which no tools exist to validate the correctness of their results because existing proof formats can only express a subset of the implemented techniques. We introduce a new proof format to express refutation proofs produced by SAT solvers that covers all existing techniques. Additionally, we implemented new tools to verify these refutation proofs.

State-of-the-art SAT solvers are used for a variety of applications. These applications rely on SAT solvers to be efficient enough to solve large problems and provide the correct results. Solvers are often used not only to find a solution for a Boolean formula, but also to make the claim that no solution exists. If a solver claims that a formula is satisfiable, we can check a reported solution linearly in the size of the formula. Yet if a solver claims that a formula has no solutions, we have to trust that the solver fully exhausted the search space for the problem. This is complicated by the fact that state-of-the-art SAT solvers employ a large array of complex techniques to maximize efficiency. Errors can be introduced at a conceptual level and an implementation level [8].

One approach to gain assurance that a SAT solver is correct is to validate the output of the SAT solver. A proof trace is a sequence of clauses that are claimed to be redundant with respect to a given formula. If a SAT solver reports

* The authors are supported by DARPA contract number N66001-10-2-4087.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 345–359, 2013.

that a given formula is unsatisfiable, it can provide a proof trace that can be checked by a smaller, easier-to-trust program called a proof checker. We only need to trust the proof checker; such a checker can validate the results of multiple solvers. Ideally, a proof trace should be compact, easy to obtain, efficient to verify, expressive enough to capture all solving techniques, and it should facilitate a simple checker implementation. We can then either trust that the proof checker is correct or go a step further and formally verify the implementation of the proof checker. By focusing our efforts on a proof checker, we gain assurance while avoiding the need to trust or formally verify a variety of solvers with differing implementations.

SAT solvers have traditionally been checked by emitting a resolution-based proof that is validated by an external checker [9,10,11]. Validating such resolution-based proofs is fast and simple; however, emitting proofs in resolution-based formats is hard and these proofs can be very large. Clausal proofs [10,12] are an alternative approach to resolution-based proofs, and are primarily checked using unit propagation. Clausal proofs are compact and easy to emit, yet verification tools based on clausal approaches are slower and more complex.

Extended Resolution (ER) [13] is the basis for some techniques used during learning [14] and preprocessing [15] in state-of-the-art SAT solvers. Refutations using ER can be exponentially smaller than refutations based solely on resolution. Examples include the pigeon-hole problems where Haken [16] showed that resolution proofs are exponential in size, while Cook [17] demonstrated how to construct polynomial-sized refutations based on ER.

The only (resolution-based) proof-checking tool that can deal with ER is tracecheck, which checks proofs emitted by the EBDDRES [18,19] solver. It simply treats ER clauses as input clauses, and thus does not verify them. Moreover, it is hard to express some techniques, such as bounded variable addition [15], using (multiple applications of) the extension rule. Other techniques, such as blocked clause addition [20], are based on a generalization of ER that cannot be expressed by conventional ER.

To overcome these problems, we propose a new clausal-proof format to compactly express techniques that go beyond resolution. Our proof format is based on a recently-introduced redundancy property of clauses called *Resolution Asymmetric Tautology* (RAT) [21]. All preprocessing and inprocessing techniques used in contemporary state-of-the-art SAT solvers can be simulated by adding and removing RAT clauses [21]. It is easy to emit a refutation in our RAT format for most techniques used in today's solvers and the proofs are compact. We present two tools to check proofs in the proposed format: a mechanically verified checker in the ACL2 theorem prover [22] and a small, fast implementation in C.

Our paper proceeds by presenting some preliminary information in Section 2. Section 3 deals with redundancy properties of clauses. We provide in Section 4 some motivating examples of why a proof format should exist that supports techniques based on ER. In Section 5, we detail resolution proofs and clausal proofs as methods to add clauses that are logically implied by a formula. Our new proof format is presented in Section 6 and two implementations of checkers

for this format are discussed in Section 7. We give an evaluation in Section 8, and we conclude in Section 9.

2 Preliminaries

We briefly review necessary background concepts: conjunctive normal form (CNF), extended resolution, and Boolean constraint propagation.

2.1 Conjunctive Normal Form

For a Boolean variable x, there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\bar{x}$. A *clause* is a finite disjunction of literals, and a CNF *formula* is a finite conjunction of clauses. A clause is a *tautology* if it contains both x and $\bar{x}$ for some variable x. The set of literals occurring in a CNF formula F is denoted by $\mathrm{LIT}(F)$. A truth assignment for a CNF formula F is a partial function τ that maps literals $l \in \mathrm{LIT}(F)$ to $\{\mathbf{t}, \mathbf{f}\}$. If $\tau(l) = v$, then $\tau(\bar{l}) = \neg v$, where $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{f} = \mathbf{t}$. Furthermore:

- A clause C is *satisfied* by assignment τ if $\tau(l) = \mathbf{t}$ for some $l \in C$.
- A clause C is *falsified* by assignment τ if $\tau(l) = \mathbf{f}$ for all $l \in C$.
- A CNF formula F is *satisfied* by assignment τ if $\tau(C) = \mathbf{t}$ for all $C \in F$.
- A CNF formula F is *falsified* by assignment τ if $\tau(C) = \mathbf{f}$ for some $C \in F$.

A CNF formula with no satisfying assignments is called *unsatisfiable*. A clause C is *logically implied* by formula F if adding C to F does not change the set of satisfying assignments of F. Two formulas are *logically equivalent* if they have the same set of solutions over the common variables. Two formulas are *satisfiability equivalent* if both have a solution or neither has a solution.

2.2 Resolution and Extended Resolution

The resolution rule states that, given two clauses $C_1 = (x \vee a_1 \vee \ldots \vee a_n)$ and $C_2 = (\bar{x} \vee b_1 \vee \ldots \vee b_m)$, the clause $C = (a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$, can be inferred by resolving on variable x. We say C is the *resolvent* of C_1 and C_2 and write $C = C_1 \bowtie C_2$. C is logically implied by any formula containing C_1 and C_2. Resolution can also be applied to sets of clauses. Let S_x be a set of clauses containing literal x and $S_{\bar{x}}$ a set of clauses containing literal $\bar{x}$. $S_x \bowtie S_{\bar{x}}$ is the set of non-tautological resolvents $R := C_1 \bowtie C_2$ with $C_1 \in S_x$ and $C_2 \in S_{\bar{x}}$.

For a given CNF formula F, the *extension rule* [13] allows one to iteratively add definitions of the form $x := a \wedge b$ by adding clauses $(x \vee \bar{a} \vee \bar{b}) \wedge (\bar{x} \vee a) \wedge (\bar{x} \vee b)$ to F, where x is a new variable and a and b are literals in the current formula. *Extended Resolution* [13] is a proof system, whereby the *extension rule* is repeatedly applied to a CNF formula F, followed by applications of the resolution rule. This proof system surpasses what can be done using only resolution; it can even polynomially simulate extended Frege systems [23].

2.3 Boolean Constraint Propagation

For a CNF formula F, *Boolean constraint propagation* (BCP) (or *unit propagation*) simplifies F based on unit clauses; that is, it repeats the following until fixpoint: If there is a unit clause $(l) \in F$, remove all clauses that contain the literal l from the set $F \setminus \{(l)\}$ and remove the literal $\bar{l}$ from all clauses in F. The resulting formula is referred to as $\mathrm{BCP}(F)$. If $(l) \in \mathrm{BCP}(F)$ for some unit clause $(l) \notin F$, we say that BCP *assigns* the literal l to $\mathbf{t}$ (and the literal $\bar{l}$ to $\mathbf{f}$). If $(l), (\bar{l}) \in \mathrm{BCP}(F)$ for some literal $l \in \mathrm{LIT}(F)$ (or, equivalently, $\emptyset \in \mathrm{BCP}(F)$), we say that BCP *derives a conflict*.

Example 1. Consider the formula $F = (a) \wedge (\bar{a} \vee b) \wedge (\bar{b} \vee c) \wedge (\bar{b} \vee \bar{c})$. We have $(a) \in F$, so $\mathrm{BCP}(F)$ removes literal $\bar{a}$, resulting in the new unit clause (b). After removal of the literals $\bar{b}$, two complementary unit clauses (c) and $(\bar{c})$ are created.

3 Verification Using the RAT Redundancy Property

Clausal proof checking relies on the addition of redundant clauses to a CNF formula. Refutations are a sequence of clauses, terminating with the empty clause, that are redundant w.r.t. a given formula. The most basic redundancy property is T (tautology). RAT is a redundancy property of clauses, computable in polynomial time, that preserves satisfiability; all preprocessing, inprocessing, and solving techniques in state-of-the-art SAT solvers can be expressed in terms of addition and removal of RAT clauses [21]. In this section, we provide an overview of redundancy properties that are covered by RAT.

For a clause C, (*asymmetric literal addition*) $\mathrm{ALA}(F, C)$ computes the *unique* clause resulting from repeating the following until fixpoint: if $l_1, \ldots, l_k \in C$ and there is a clause $(l_1 \vee \cdots \vee l_k \vee l) \in F \setminus \{C\}$ for some literal l, let $C := C \cup \{\bar{l}\}$. A clause C has property AT (asymmetric tautology) with respect to a CNF formula F if and only if $\mathrm{ALA}(F, C)$ has property T. Clauses with the property AT are also known as *reverse unit propagation* (RUP) clauses [10,12]. A clause C is RUP (or has AT) if unit propagation on an assignment that falsifies C will result in a conflict. More formally, let $\overline{C}$ denote the set of unit clauses that falsify all literals in C. Clause C is RUP if and only if $\emptyset \in \mathrm{BCP}(F \cup \overline{C})$.

Given a CNF formula F and a clause $C \in F$, C *has property* R$\mathcal{P}$ (with $\mathcal{P} \in \{\mathrm{T}, \mathrm{AT}\}$) if and only if either (i) C has the property $\mathcal{P}$, or (ii) there is a literal $l \in C$ such that for each clause $C' \in F$ with $\bar{l} \in C'$, each resolvent in $C \bowtie C'$ has $\mathcal{P}$. In the latter case, we say C has R$\mathcal{P}$ on l. Clauses with property RT (resolution tautology) with respect to a CNF formula F are also known as *blocked clauses* [20].

If a clause C has one of the redundancy properties w.r.t. a CNF formula F, one can add C to F and preserve satisfiability, or remove C from F and preserve unsatisfiability. We will focus on adding redundant clauses to a given formula either with the redundancy property AT, which is the strongest redundancy property that preserves logical equivalence, or RAT, which is the strongest redundancy property that preserves satisfiability equivalence. Fig. 1 shows the relationships between clause redundancy properties.

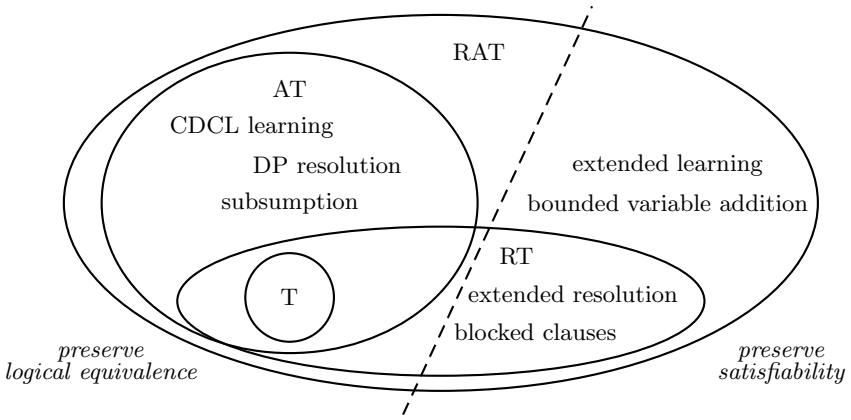


Fig. 1. Relationships between clause redundancy properties that can be computed in polynomial time. Techniques shown in an area denote the cheapest check one can apply to verify a proof trace from a SAT solver that uses that technique. All techniques used in state-of-the-art SAT solvers can expressed as a sequence of RAT clauses [21]. The dashed line separates techniques that preserve logical equivalence and those that preserve satisfiability.

Example 2. Let formula $F = (a \vee b) \wedge (b \vee c) \wedge (\bar{b} \vee \bar{c})$.

- The clause $(a \vee \bar{a})$ is a tautology because it contains a and $\bar{a}$ and therefore has T and thus AT, RT and RAT.
- The clause $(a \vee \bar{c})$ does not have T. However, it has RT (and RAT) with respect to F on literal a, because F contains no clauses with literal $\bar{a}$. Furthermore, it also has AT because unit propagation under the assignment $(\bar{a} \wedge c)$ results in a conflict.
- The clause $(\bar{a} \vee c)$ has RAT, but not T, AT, or RT. It is clear that $(\bar{a} \vee c)$ does not have T. Unit propagation under the assignment $(a \wedge \bar{c})$ does not result in a conflict, so $(\bar{a} \vee c)$ does not have AT. Also, $(\bar{a} \vee c)$ does not have RT, because there is a non-tautological resolvent on $\bar{a}$ with $(a \vee b)$ and on c with $(\bar{b} \vee \bar{c})$. Finally, there is only one resolvent on literal $\bar{a}$. The resolvent of $(\bar{a} \vee c)$ and $(a \vee b)$ is $(b \vee c)$, which is already in F. Therefore, unit propagation on the assignment $(\bar{b} \wedge \bar{c})$ will result in a conflict. Hence, $(\bar{a} \vee c)$ has RAT.

4 Extended Resolution in Practice

This section provides an overview of several techniques that use Extended Resolution or a generalization. We will present these techniques as motivating examples for our proof format (discussed in Section 6) based on RAT clauses.

4.1 Manually-Constructed Proofs

A classic problem known to be hard for resolution provers is the *pigeon-hole* problem. A pigeon-hole problem of size n (denoted by PH_n) describes whether

n pigeons can be placed in $n-1$ holes such that each hole contains at most one pigeon. Although the problem is easy for any n from an abstract level, it is hard to refute the straight-forward translation of pigeon-hole problems into a CNF formula. The number of resolutions to derive the empty clause is exponential in n, and SAT solvers also require exponential runtime on these problems.

Let Boolean variable $x_{i,j}$ denote whether pigeon i is in hole j with $i \in \{1..n\}$ and $j \in \{1..n{-}1\}$. The straight-forward SAT translation of PH_n consists of a set of clauses describing that pigeon i is in at least one hole, and a set of clauses enforcing that if pigeon i is in hole j that pigeon $k > i$ cannot be in hole j.

$$\bigwedge_{i\in\{1..n\}} (x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,n-1}) \wedge \bigwedge_{i,j\in\{1..n-1\}} \bigwedge_{k\in\{i+1..n\}} (\bar{x}_{i,j} \vee \bar{x}_{k,j}) \quad (1)$$

Although the problem is hard for resolution [16], polynomial-size refutations do exist for the Extended Resolution [13] technique. ER can reduce PH_n into PH_{n-1}. Applying the reduction $n{-}1$ times, results in the trivial PH_1. The first step of the reduction is introducing auxiliary Boolean variables $y_{i,j}$ with $i \in \{1..n{-}1\}$ and $j \in \{1..n{-}2\}$. When all reduction steps are applied, these variables $y_{i,j}$ encode that pigeon i is in hole j in the PH_{n-1} problem. Let

$$y_{i,j} := x_{i,j} \vee (x_{n,j} \wedge x_{i,n-1}) \quad (2)$$

The definition can be translated into clauses that have all RT on $y_{i,j}$.

$$(y_{i,j} \vee \bar{x}_{i,j}) \wedge (y_{i,j} \vee \bar{x}_{n,j} \vee \bar{x}_{i,n-1}) \wedge (\bar{y}_{i,j} \vee x_{i,j} \vee x_{n,j}) \wedge (\bar{y}_{i,j} \vee x_{i,j} \vee x_{i,n-1}) \quad (3)$$

By adding the clauses (3) with $i \in \{1..n-1\}, j \in \{1..n-2\}$ to the formula, the clauses encoding the left set of clauses of (1) with $y_{i,j}$ variables are:

$$(y_{i,1} \vee y_{i,2} \vee \cdots \vee y_{i,n-2}) \quad (4)$$

Notice that the clauses (4) have AT after the presence of (3): First assign $y_{i,1}, y_{i,2}, \ldots, y_{i,n-2}$ to false, then BCP assigns $x_{i,1}, x_{i,2}, \ldots, x_{i,n-2}$ to false using $(y_{i,j} \vee \bar{x}_{i,j})$. This in turn makes $x_{i,n-1}$ true by $(x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,n-1})$. After these assignments, all the clauses $(y_{i,j} \vee \bar{x}_{n,j} \vee \bar{x}_{i,n-1})$ become unit assigning all $x_{n,1}, x_{n,2}, \ldots, x_{n,n-2}$ to false. Now, $(x_{n,1} \vee x_{n,2} \vee \cdots \vee x_{n,n-1})$ assigns $x_{n,n-1}$ to true, and a conflict arises because the clause $(\bar{x}_{i,n-1} \vee \bar{x}_{n,n-1})$ is falsified.

The right set of clauses of (1) using $y_{i,j}$ variables is required to finish the reduction. These $(\bar{y}_{i,j} \vee \bar{y}_{k,j})$ don't have RAT. Yet the clauses $(\bar{y}_{i,j} \vee \bar{y}_{k,j} \vee x_{i,n-1})$ have AT and in the presence of $(\bar{y}_{i,j} \vee \bar{y}_{k,j} \vee x_{i,n-1})$, $(\bar{y}_{i,j} \vee \bar{y}_{k,j})$ has AT as well.

$$(\bar{y}_{i,j} \vee \bar{y}_{k,j} \vee x_{i,n-1}); \quad (\bar{y}_{i,j} \vee \bar{y}_{k,j}) \quad (5)$$

So by adding the clauses (3), (4), and (5) —all having RT or AT, and thus RAT— we can reduce the PH_n problem into a PH_{n-1} problem. Repeating this $n-1$ times results in a refutation. A similar, but much larger proof of unsatisfiability can be obtained by combining resolution and ER, as in [17].

4.2 Extended Learning

Our manually-constructed extended resolution proof above illustrates the potential of introducing new variables; however, it is hard to capitalize on this potential in practice. Most applications of the extension rule will result in useless variables which can slow down the search. The most serious study of practical ER during search looks for a pattern between consecutive conflict clauses [14]. When such a pattern is found, a new variable is introduced.

The pattern consists of conflict clauses that differ in exactly one literal. Given two successive conflict clauses $C = (l_1 \vee \alpha)$ and $D = (l_2 \vee \alpha)$ with α being a disjunction of literals, the extension rule is applied using $z := (\bar{l}_1 \vee \bar{l}_2)$. Newly introduced variables are used to shorten future conflict clauses. Let γ be a disjunction of literals. If a conflict clause $(l_1 \vee l_2 \vee \gamma)$ is found and the variable $z := (l_1 \vee l_2)$ was created in the past, $(z \vee \gamma)$ is added to the learned clause database.

4.3 Bounded Variable Addition

One of the most effective preprocessing/inprocessing techniques is *Bounded Variable Elimination* (BVE) [24]. This technique tries to reduce the sum of the number of variables and the number of clauses by eliminating variables. Given a CNF formula F, let F_l denote the subset of clauses of F that contains literal l. BVE searches for a variable x, such that it can replace F_x and $F_{\bar{x}}$ by the set of non-tautological resolvents of $F_x \bowtie F_{\bar{x}}$ if and only if $|F_x \bowtie F_{\bar{x}}| \leq |F_x| + |F_{\bar{x}}|$.

Recently a complementary technique was proposed, called *Bounded Variable Addition* (BVA) [15], that introduces new variables. BVA uses the same metric for substitution: new variables are added while the sum of the number of variables and clauses strictly decreases. BVE is based on resolution and therefore one can use existing resolution and clausal proof formats to verify an implementation. However, BVA cannot be simulated by resolution and simulation by ER is non-trivial. Yet a proof trace of BVA can elegantly be expressed using RAT clauses.

BVA works as follows. Given a CNF formula F and a new Boolean variable x, BVA searches for sets of clauses G_x (clauses containing literal x) and $G_{\bar{x}}$ (clauses containing literal $\bar{x}$), such that all non-tautological resolvents of $G_x \bowtie G_{\bar{x}}$ are in F and $|G_x \bowtie G_{\bar{x}}| > |G_x| + |G_{\bar{x}}|$. Whenever BVA finds such a G_x and $G_{\bar{x}}$, it replaces the resolvents by: $F := (F \cup G_x \cup G_{\bar{x}}) \setminus G_x \bowtie G_{\bar{x}}$.

Example 3. Consider the formula $F = (a \vee c) \wedge (a \vee d) \wedge (a \vee e) \wedge (b \vee c) \wedge (b \vee d) \wedge (b \vee e)$. For F there exists $G_x = (a \vee x) \wedge (b \vee x)$ and $G_{\bar{x}} = (\bar{x} \vee c) \wedge (\bar{x} \vee d) \wedge (\bar{x} \vee e)$ such that all non-tautological resolvents of $G_x \bowtie G_{\bar{x}}$ are in F and $|G_x \bowtie G_{\bar{x}}| > |G_x| + |G_{\bar{x}}|$.

All clauses added by the extension rule are blocked (have RT) on the new variable. However, this is not the case with BVA. In Example 3, none of the clauses in G_x and $G_{\bar{x}}$ are blocked; all these clauses have RAT on the new variable. Without loss of generality consider a clause $C \in G_x$. All resolvents $R := C \bowtie C'$ with $C' \in G_{\bar{x}}$ are either tautologies or R is subsumed by F (namely, $R \in F$). Both tautologies and subsumptions are asymmetric tautologies, and therefore, C has RT.

5 Existing Proof Formats

Conflict-driven clause learning (CDCL) [25] is the leading paradigm of modern SAT solvers. A core aspect of CDCL solvers is the addition and removal of clauses. The main form of reasoning is known as conflict analysis, which adds clauses. Additionally, state-of-the-art CDCL solvers use preprocessing and inprocessing techniques that both add and remove clauses. Proof formats for CDCL solvers express *how to check* that a clause addition step preserves satisfiability. This section provides an overview of existing formats. In next section, we present our new format.

We appeal to the notion that *lemmas* are used to construct a proof of a theorem. Here, lemmas represent the learned clauses and the theorem is the statement that the formula is unsatisfiable. From now on, we will use the term clauses to refer to input clauses, while lemmas will refer to added clauses.

5.1 Resolution Proofs

The early approaches to prove refutations produced by SAT solvers were based on resolution [9]. The lemmas computed by CDCL solvers can be simulated by a sequence of resolutions [26]. Let L be a lemma and $\{C_1, \ldots, C_n\}$ be the input clauses. For each L, one must specify a sequence such that $L = (((C_i \bowtie C_j) \ldots) \bowtie C_k)$. This sequence may use added lemmas to construct new lemmas.

As resolution is an elementary operation, simple and fast checking algorithms exist [9,10,11]. However, resolution proofs can be huge (dozens of gigabytes). It may also be hard to modify a SAT solver to emit a resolution refutation; for instance, one must determine the clauses on which to apply resolution, and specifying the order may be difficult. Since only a handful of (mostly outdated) solvers support resolution-based proofs, a user would need to modify a solver to emit resolution proofs. Even for the author(s) of a SAT solver this is not an easy task. In the case one wishes to integrate a portfolio of SAT solvers into a SMT solver or theorem prover, it would be a daunting task to enhance them all to emit resolution proofs.

5.2 Clausal Proofs

An alternative approach using *clausal proofs* was proposed by Goldberg and Novikov [12]. They observed that each lemma L learned by CDCL conflict analysis can be checked using BCP. Lemmas, like clauses, are disjunctions of literals. If $\text{BCP}(F \cup \overline{L})$ results in a conflict, i.e., produces the empty clause $\emptyset$, then L is implied by F. Notice that this corresponds to the redundancy property AT. Lemmas with AT are also known as *reverse unit propagation* (RUP) lemmas [10].

Clausal proofs are represented as a queue of lemmas $(L_1, \ldots, L_m)$ such that $L_m = \emptyset$. Given a CNF formula F, a clausal proof of F consists of lemmas L_i that are redundant w.r.t. F. Let $F_0 = F$ and $F_i := F_{i-1} \cup \{L_i\}$. Existing clausal proof formats expect that lemma L_i has AT w.r.t. F_{i-1}. In our proposed format (see Section 6) lemmas should have the more general RAT redundancy property.

The elegance of clausal proofs is that they can be expressed in conjunctive normal form; however, the order of the lemmas is important. Clausal proofs are significantly smaller when compared to resolution proofs, and only minor modifications of a SAT solver are required to output such proof records. However, checking of clausal proofs can be quite expensive. Checking algorithms for clausal proofs are also typically more complex than those for resolution proofs, making it harder to trust or prove correctness of the algorithm.

```
RUPchecker (CNF formula F, queue Q of lemmas)
1   while Q is not empty
2     L := Q.pop()
3     F' := BCP(F ∪ ¬L)
4     if ∅ ∉ F' then return "checking failed"
5     F := BCP(F ∪ L)
6     if ∅ ∈ F then return "unsatisfiable"
7   return "all lemmas validated"

BCP (CNF formula F)
11  while ∃ (x) ∈ F do
12    for C ∈ F with x̄ ∈ C do
13      C := C \ {x̄}
14    for C ∈ F with x ∈ C do
15      F := F \ {C}
16  return F
```

Fig. 2. Pseudo-code to check clausal proofs for lemmas with AT (or RUP lemmas)

Fig. 2 shows the pseudo-code of a clausal, proof-checking algorithm for lemmas with AT. The input is a CNF formula F and a queue Q of lemmas representing a refutation of F. Lemmas are sorted in chronological order as learned by the SAT solver. While Q is not empty (line 1), its front lemma L is popped (line 2). If unit propagation on F using $\overline{L}$ does not derive a conflict, then we fail to check that L is logically implied by F and terminate (line 3 and 4). Otherwise, L is added to F. In case L was unit, the new F is simplified using BCP (line 5). If unit propagation results in a conflict, a top-level contradiction is found, meaning that the formula is unsatisfiable (line 6). If the algorithm reaches the end (line 7), all lemmas in Q were validated but no top-level conflict was encountered.

6 The RAT Proof Format

In this section, we propose the new RAT proof format. This is an alternative clausal-proof format that supports both AT (or RUP) and RAT lemmas.

The main decision regarding a proof format for ER and its generalizations was whether to use a resolution-style or clausal-style proof format. Apart from the known disadvantages of resolution-style proofs (recall Section 5.1), there is another drawback of ER proofs: techniques like blocked clause addition [20], cannot be expressed using the extension rule. Consequently, if one wants to verify all known techniques, a clausal-style proof format seems the most viable option.

We considered whether to specify the simplest redundancy property for each lemma. All redundancy properties are covered by RAT, so it is not necessary to distinguish between them; however, efficiency may be gained by distinguishing

them. In practice, the majority of lemmas has the AT property; therefore, by first checking for AT, which is part of the RAT check, we reduce overhead.

In case a lemma does not have AT, our proof format expects the lemma to have RAT on its first literal. Fig. 3 shows three refutations in the RUP (mid left) and RAT formats (both on the right). The last RAT refutation shows that one can introduce new variables in a RAT proof — which is not allowed in RUP proofs.

CNF formula

```
p cnf 4 16
 1  2  3  4 0
 1  2  3 -4 0
 1  2 -3  4 0
 1  2 -3 -4 0
 1 -2  3  4 0
 1 -2  3 -4 0
 1 -2 -3  4 0
 1 -2 -3 -4 0
-1  2  3  4 0
-1  2  3 -4 0
-1  2 -3  4 0
-1  2 -3 -4 0
-1 -2  3  4 0
-1 -2  3 -4 0
-1 -2 -3  4 0
-1 -2 -3 -4 0
```

smallest RUP proof

```
1 2 3 0
1 2 0
1 3 0
1 0
2 3 0
2 0
3 0
0
```

smallest RAT proof

```
1 0
2 0
3 0
0
```

RAT proof with ER

```
 5  1  2  0
 5  1 -2  0
 5 -1  2  0
 5 -1 -2  0
-5  3  4  0
-5  3 -4  0
-5 -3  4  0
-5 -3 -4  0
 5  1  0
 5  0
 3  0
 0
```

Fig. 3. An example of a CNF problem in the typical DIMACS format (left) as well as three refutations; one in the RUP format (mid left) and two in RAT format (right). The proofs in the middle show a smallest proof (in the number of lemmas) for RUP and RAT. Whitespaces can be of any length; the spacing is to improve readability. A `0` marks the end of clauses and lemmas. The RUP and RAT formats have the same syntax. Only the RAT format allows lemmas to have the RAT redundancy property.

7 Implementation

We have implemented[1] a RAT checker in ACL2 that is concise in its expression, and, more importantly, mechanically verified. Our proof of correctness for our RAT checker hinges on the mechanical proof of the redundancy of the RAT property presented in Section 3. We did this by modeling the RAT proof-checking algorithm as an ACL2 function, and then we used the ACL2 mechanical proof-checking system to assure that our RAT proof-checking algorithm is valid.

[1] The material presented in the paper, such as the formal proof, our tools, and the used benchmarks are available on `www.cs.utexas.edu/~marijn/rat/`. Our proof contains roughly 150 ACL2 (definitions and proof request) events.

Apart from our mechanically verified checker, we implemented a concise RAT checker in C (about 200 lines of code). Fig. 4 shows its pseudo-code. Our RAT checker extends the RUP checker pseudo-code (recall Fig. 2) and uses the same input parameters, initialization (lines 1 and 2), and termination (lines 8 to 10). Before validating a lemma L, it first checks whether L has AT (line 3). Otherwise, the expensive RAT check is applied. According to the RAT proof format, lemma L should have RAT on its first literal l (line 4). We compute for all clauses $C' \in F_{\bar{l}}$ (line 5), the resolvent $R := C' \bowtie L$ (line 6), and check whether R has AT (line 7). If all these R have AT, L is added to F (line 8); otherwise, we return "checking failed".

```
   RATchecker (CNF formula F, queue Q of lemmas)
1    while Q is not empty
2      L := Q.pop()
3      if ∅ ∉ BCP(F ∪ L̄) then          // check if L has AT, otherwise
4        let l be the first literal in L.      // assume L has RAT on l
5        forall C' ∈ F_l̄ do
6          R := C' ⋈ L
7          if ∅ ∉ BCP(F ∪ R̄) then return "checking failed"
8      F := BCP(F ∪ L)
9      if ∅ ∈ F then return "unsatisfiable"
10   return "all lemmas validated"
```

Fig. 4. Pseudo-code to check Resolution Asymmetric Tautology (RAT) proofs

In general, the RAT check (lines 4 to 7) is more expensive than the AT check (line 3), because one has to do the AT check for each R. However, in practice, for half the RAT lemmas, $F_{\bar{l}}$ is empty and we skip lines 4 to 7.

The main reason why a RAT checker is more complex and less efficient, as compared to a RUP checker, is the requirement in line 5 to compute $F_{\bar{l}}$, the set of clauses containing literal $\bar{l}$. In order to do this computation efficiently, the checker needs to maintain a full occurrence list of all clauses. Alternatively, a RUP checker could use a watch-pointer data structure.

Notice that for all checks (lines 3 and 7), all literals $l' \in L \setminus l$ will be assigned to false. One can optimize a RAT checker by first assigning all the literals $l' \in L \setminus l$ to false followed by unit propagation and perform the checks on this assignment. This optimization is implemented in our C checker.

8 Evaluation

To demonstrate the usefulness of the RAT proof format, we experimented with our tools on the problems discussed in Section 4. We ran our tests on a 4-core Intel Xeon CPU E31280 3.50GHz, 32 Gb RAM machine running Ubuntu 10.04.

8.1 Manually-Constructed Proofs

The first experiment evaluates the performance of our RAT checking tools on the manually-contructed proofs of PH_n problems presented in Section 4.1. Although these problems are notoriously hard for SAT solvers, the manually-contructed proofs are small and can be checked in a fraction of a second using our C implementation, see Table 1. The ACL2 checker is significantly slower, but was not written with speed in mind.

Table 1. Evaluation of manually-constructed proofs of PH_n problems. The first column shows the benchmark name. The next two columns show the number of variables in the input formula and in the proof. The number of original, AT, and RAT clauses, as well as their sum (total) is shown in the next four columns. The last two columns show the time (in seconds) to check the proofs using our C and ACL2 implementations.

	#variables		*#clauses*				*time*	
benchmark	input	proof	input	AT	RT	total	C	ACL2
PH_6	30	70	81	160	145	386	0.003	0.26
PH_7	42	112	133	280	301	714	0.005	1.55
PH_8	56	168	204	448	560	1,212	0.007	7.91
PH_9	72	240	297	672	960	1,929	0.010	34.26
PH_{10}	90	330	415	960	1,545	2,920	0.014	129.78
PH_{11}	110	440	561	1,320	2,365	4,246	0.016	440.69
PH_{12}	132	572	738	1,760	3,476	5,974	0.020	1358.65

8.2 Extended Learning

We modified the solver `GlucosER 1.0` [14], which combines CDCL learning and ER, such that it emits a proof in the proposed RAT format[2]. We evaluated the *C* checking tool on benchmarks where `GlucosER` has an edge over SAT solvers without ER learning, such as the PH_n instances and the Urquhart benchmarks [27].

Table 2 shows the results of the second experiment with our C checking tool. Compared to the prior results, the C tool requires much more time to verify the output of `GlucosER`. Notice that although `GlucosER` uses ER, it cannot compete with the manually-constructed proofs on the same problems.

8.3 Bounded Variable Addition

The technique BVA, discussed in Section 4.3, is a helpful preprocessing technique for several families of benchmarks, including the PH_n problems and some hard bioinformatics [28] benchmarks. We modified a preprocessing tool which includes a BVA implementation, `coprocessor` [29], and the `Glucose 2.1` solver [30] (the winner of the SAT 2012 Challenge) to output lemmas in the RAT format. We verified the merged file consisting of the original problem and the lemmas produced by the preprocessor and solver.

[2] Additionally, we removed the code that allows reuse of variables that have been eliminated. The removal of this part of the code made it easier to verify and has no noticeable effect on the performance.

Table 2. Evaluation of Extended Learning on PH_n and Urquhart benchmarks. The first column shows the benchmark name. The next two columns show the number of variables in the input formula and in the proof. The number of original, AT, and RAT clauses, as well their sum (total) is shown in the next four columns. The last two columns show the time (in seconds) to solve the benchmarks and to check the emitted proofs using our C checker.

	#variables		*#clauses*				*time*	
benchmark	input	proof	input	AT	RT	total	solving	checking
PH_{10}	90	379	415	99,682	867	100,973	5.28	24.72
PH_{11}	110	814	561	260,677	2,112	263,350	13.51	72.08
PH_{12}	132	1,450	738	1,512,453	3,954	1,517,145	145.29	3,521.23
Urq_3_5	45	2,126	446	281,761	6,243	288,450	8.33	17.38
Urq_3_6	54	3,842	688	1,156,477	11,364	1,168,529	52.69	152.36
Urq_3_7	42	1,147	342	102,950	3,315	106,607	2.20	3.95
Urq_3_8	44	1,518	416	149,286	4,422	154,124	3.70	5.86

Table 3 shows the results regarding the performance improvements due to BVA and the RAT proof checking costs. The performance difference when using `Glucose 2.1` on the original and BVA-preprocessed instances is huge: the largest instance cannot be solved in 12 hours, while the preprocessed formula is solved in two minutes. It is important to check that these gains are not caused by a bug. Our proof checker confirms that the refutation is correct.

Table 3. Evaluation of checking RAT produced by BVA preprocessing on PH_n and bioinformatics (rbclY) benchmarks. The timeout (denoted by —) is 12 hours. The first column shows the benchmark name. The next three columns show the number of variables, the number of clauses, and the solving time (in seconds) of the original formula. The next three columns show the same information for the BVA preprocessed formula. The last three columns show the number of AT and RAT clauses in the proofs, as well as the time (in seconds) to check the proofs using our C checker.

	original			BVA preprocessed			RAT proof checking		
benchmark	*#vars*	*#cls*	*time*	*#vars*	*#cls*	*time*	#AT	#RAT	*time*
PH_{10}	90	330	7.71	117	226	1.25	42,853	198	4.19
PH_{11}	110	440	84.42	151	281	12.34	225,959	295	152.82
PH_{12}	132	572	494.29	187	342	8.45	181,603	402	69.01
rbcl_07	1,128	57,446	52.92	1,784	7,598	2.88	72,073	19,681	6.76
rbcl_08	1,278	67,720	1,763.36	1,980	9,004	10.72	151,894	22,830	37.58
rbcl_09	1,430	79,118	—	2,190	10,492	129.20	882,213	26,639	2,631.28

We are working on techniques to decrease the time to check RAT proofs. Initial results indicate that RAT verification can be realized in a time similar to the solving time. Improvement to the speed will likely increase the complexity of the checker implementation.

9 Conclusions

We presented a new clausal proof format for SAT solvers. The crucial difference is that we allow lemmas to have the redundancy property RAT. Since all techniques used in state-of-the-art SAT solvers can be simulated by the addition and removal of RAT lemmas [21], our new format facilitates the verification of results produced by SAT solvers. For most techniques, it is easy to modify a solver to emit a proof in our format, which includes CDCL and ER learning, and bounded variable addition.

Two major challenges remain to conveniently verify the results of SAT solvers. Our C implementation may be slow when a solver emits a huge proof. It is still an open question whether only minor modifications to SAT solvers are needed for all techniques. For example, Gaussian elimination of XOR constraints can be simulated using ER [18,19] techniques, but these methods require several modifications for SAT solvers that use Gaussian elimination.

Our new format and our tools are the first, complete approach toward SAT solver verification. We expect them to be used to check implementations of the more complex techniques, in particular those based on ER. Our new format and tools support the development of new techniques that may further capitalize on the strength of ER.

References

1. Goldberg, E.I., Prasad, M.R., Brayton, R.K.: Using SAT for combinational equivalence checking. In: DATE, pp. 114–121 (2001)
2. Mishchenko, A., Chatterjee, S., Brayton, R.K., Eén, N.: Improvements to combinational equivalence checking. In: Hassoun, S. (ed.) ICCAD, pp. 836–843. ACM (2006)
3. Baumgartner, J., Mony, H., Paruthi, V., Kanzelman, R., Janssen, G.: Scalable sequential equivalence checking across arbitrary design transformations. In: ICCD. IEEE (2006)
4. Kaiss, D., Skaba, M., Hanna, Z., Khasidashvili, Z.: Industrial strength SAT-based alignability algorithm for hardware equivalence verification. In: FMCAD, pp. 20–26. IEEE Computer Society (2007)
5. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of bdds. In: DAC, pp. 317–320 (1999)
6. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
7. Chen, Y., Safarpour, S., Marques-Silva, J.P., Veneris, A.G.: Automated design debugging with maximum satisfiability. IEEE Trans. on CAD of Integrated Circuits and Systems 29(11), 1804–1817 (2010)
8. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 44–57. Springer, Heidelberg (2010)
9. Zhang, L., Malik, S.: Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE, pp. 10880–10885 (2003)

10. Van Gelder, A.: Verifying rup proofs of propositional unsatisfiability. In: ISAIM (2008)
11. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 260–274. Springer, Heidelberg (2010)
12. Goldberg, E.I., Novikov, Y.: Verification of proofs of unsatisfiability for cnf formulas. In: DATE, pp. 10886–10891 (2003)
13. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning 2, pp. 466–483. Springer (1983)
14. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In: Fox, M., Poole, D. (eds.) AAAI. AAAI Press (2010)
15. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of boolean formulas. In: Proceedings of Haifa Verification Conference 2012 (2012)
16. Haken, A.: The intractability of resolution. Theor. Comput. Sci. 39, 297–308 (1985)
17. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News 8(4), 28–32 (1976)
18. Sinz, C., Biere, A.: Extended resolution proofs for conjoining bdds. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, pp. 600–611. Springer, Heidelberg (2006)
19. Jussila, T., Sinz, C., Biere, A.: Extended resolution proofs for symbolic SAT solving with quantification. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 54–60. Springer, Heidelberg (2006)
20. Kullmann, O.: On a generalization of extended resolution. Discrete Applied Mathematics 96-97, 149–176 (1999)
21. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 355–370. Springer, Heidelberg (2012)
22. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Boston (2000)
23. Cook, S.A., Reckhow, R.A.: The relative efficiency of propositional proof systems. The Journal of Symbolic Logic 44(1), 36–50 (1979)
24. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
25. Marques-Silva, J.P., Lynce, I., Malik, S.: 4. In: Conflict-Driven Clause Learning SAT Solvers. Handbook of Satisfiability, pp. 131–153. IOS Press (February 2009)
26. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. JAIR 22, 319–351 (2004)
27. Urquhart, A.: Hard examples for resolution. J. ACM 34(1), 209–219 (1987)
28. Bonet, M.L., John, K.S.: Efficiently calculating evolutionary tree measures using SAT. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 4–17. Springer, Heidelberg (2009)
29. Manthey, N.: Coprocessor 2.0 – A flexible CNF simplifier. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 436–441. Springer, Heidelberg (2012)
30. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Boutilier, C. (ed.) IJCAI, pp. 399–404 (2009)

Hierarchical Reasoning and Model Generation for the Verification of Parametric Hybrid Systems

Viorica Sofronie-Stokkermans

University of Koblenz-Landau and Max-Planck-Institut für Informatik, Saarbrücken
sofronie@uni-koblenz.de

Abstract. In this paper we study possibilities of using hierarchical reasoning, quantifier elimination and model generation for the verification of parametric hybrid systems, where the parameters can be constants or functions. Our goal is to automatically provide guarantees that such systems satisfy certain safety or invariance conditions. We first analyze the possibility of automatically generating such guarantees in the form of constraints on parameters, then show that we can also synthesise so-called criticality functions, typically used for proving stability and/or safety of hybrid systems. We illustrate our methods on several examples.

1 Introduction

In this paper we study possibilities of using hierarchical reasoning, quantifier elimination and model generation for the analysis and verification of increasingly more general classes of parametric hybrid systems (where the parameters can be either constants or functions, possibly modeling parametric "updates" of the system and/or environment). Our goal is to give methods which allow to automatically provide guarantees that such systems satisfy certain safety or invariance conditions. Our main contributions can be described as follows:

- We refine the methods for generating constraints on parameters which we gave in [15,3,4] and apply them to increasingly more complex hybrid systems (including non-linear hybrid systems or families of hybrid systems).
- We study possibilities of automatically generating criticality functions (used to characterize regions where safety properties can be guaranteed).

There exist approaches to the verification of parametric hybrid automata (e.g. by Henzinger et al. [1], Frehse [6], Wang [17], Tiwari et al. [7] and Cimatti et al. [2], Platzer et al. [10,11]) possibly used for synthesis [16]. However, in most cases only situations in which the parameters are constants were considered. In [15] and [3,4] we made first steps towards the verification of hybrid systems with a more complex form of parametricity, but with a focus on linear hybrid automata. We here extend these results to parametric hybrid automata, and also explore possibilities of generating criticality functions in order to give guarantees for safety in such systems. The main advantage of our method for generating

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 360–376, 2013.

criticality functions compared with existing approaches based on Lyapunov function computation (cf. e.g. [12]) is that we restrict to very simple properties of the continuous variables (such as monotonicity properties) and thus avoid the complexity introduced by the complicated dynamics.

Structure. Section 2 introduces the main definitions and results on local theory extensions and hybrid automata used in the paper. In Section 3 we present a general result on generating constraints on parameters under which certain formulae are invariant (which we apply for the verification of parametric linear hybrid automata, then extend to parametric hybrid automata and systems thereof). In Section 4 we present a method for generating criticality functions.

2 Preliminaries

We present the main definitions and results on local theory extensions and hybrid automata used in the paper. For details on local theory extensions we refer to [13,8,9]; for details on hybrid automata we refer for instance to [1].

Local Theory Extensions. Let $\mathcal{T}_0$ be a theory with signature $\Pi_0 = (S, \Sigma_0, \mathsf{Pred})$, where S is a set of sorts, Σ_0 a set of function symbols, and Pred a set of predicate symbols. We consider extensions $\mathcal{T}_1$ of $\mathcal{T}_0$ with new function symbols in a set Σ_1 whose properties are axiomatized with a set $\mathcal{K}$ of clauses[1] whose free variables are considered to be universally quantified (Notation: $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$, where $\mathcal{T}_1$ has signature $\Pi = (S, \Sigma_0 \cup \Sigma_1, \mathsf{Pred})$ - also written $\Pi = \Pi_0 \cup \Sigma_1$). An extension $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ is *local* [13] if it satisfies condition (Loc):[2]

(Loc) For every finite set G of ground clauses $\mathcal{T}_1 \cup G \models \bot$ iff $\mathcal{T}_0 \cup \mathcal{K}[G] \cup G \models \bot$

where $\mathcal{K}[G]$ is the set of instances of $\mathcal{K}$ where all terms starting with an extension function are in the set $\mathsf{st}(\mathcal{K}, G)$ of all ground terms occurring in $\mathcal{K}$ or G.

A similar *extended locality condition* (ELoc) [9] can be defined for theory extensions in which $\mathcal{K}$ consists of augmented clauses ((ELoc) states that $\mathcal{T}_1 \cup \Gamma \models \bot$ iff $\mathcal{T}_0 \cup \mathcal{K}[\Gamma] \cup \Gamma \models \bot$ holds for all Σ_0-*extended ground clauses* $\Gamma = \Gamma_0 \cup G$, where Γ_0 is a Π_0^c-sentence and G is a finite set of ground Π^c-clauses). In [8] we introduced Ψ-(extended) locality conditions (Loc^Ψ) and (ELoc^Ψ), where Ψ is a closure operator on ground terms, in which the instances of $\mathcal{K}$ to be considered without loss of completeness are $\mathcal{K}[\Psi_\mathcal{K}(G)]$ instead of $\mathcal{K}[G]$, where $\Psi_\mathcal{K}(G) = \Psi(\mathsf{st}(\mathcal{K}, G))$.

Hierarchical reasoning. Consider a Ψ-local theory extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$. Condition ($\mathsf{Loc}^\Psi$) requires that, for every set G of ground Π^c clauses, $\mathcal{T}_0 \cup \mathcal{K} \cup G \models \bot$ iff $\mathcal{T}_0 \cup \mathcal{K}[\Psi_\mathcal{K}(G)] \cup G \models \bot$. In all clauses in $\mathcal{K}[\Psi_\mathcal{K}(G)] \cup G$ the function symbols in Σ_1 have as arguments only ground terms, so $\mathcal{K}[\Psi_\mathcal{K}(G)] \cup G$ can be flattened

[1] We can also consider sets $\mathcal{K}$ of *augmented clauses*, i.e. of axioms of the form $(\Phi(x_1, \ldots, x_n) \vee C(x_1, \ldots, x_n))$, where $\Phi(x_1, \ldots, x_n)$ is an *arbitrary first-order formula* in the base signature Π_0 and $C(x_1, \ldots, x_n)$ is a *clause* containing Σ_1-functions.

[2] In what follows, when we refer to *sets* G of ground clauses we assume that they are in the signature $\Pi^c = (S, \Sigma \cup \mathcal{C}, \mathsf{Pred})$, where $\mathcal{C}$ is a set of new constants.

and purified[3]. The set of clauses thus obtained has the form $\mathcal{K}_0 \cup G_0 \cup \mathsf{Def}$, where Def consists of ground unit clauses of the form $f(c_1, \ldots, c_n) = c$, where $f \in \Sigma_1$, $c, c_1, \ldots, c_n$ are constants, and $\mathcal{K}_0$ and G_0 do not contain Σ_1-function symbols.

Theorem 1 ([13,8]). *Let $\mathcal{K}$ be a set of clauses. Assume that $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ is a Ψ-local theory extension. For any set G of ground clauses, let $\mathcal{K}_0 \cup G_0 \cup \mathsf{Def}$ be obtained from $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ by flattening and purification, as explained above. Then the following are equivalent to $\mathcal{T}_1 \cup G \models \bot$:*

(1) $\mathcal{T}_0 \cup \mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G \models \bot$.

(2) $\mathcal{T}_0 \cup \mathcal{K}_0 \cup G_0 \cup \mathsf{Con}_0 \models \bot$, *where* $\mathsf{Con}_0 = \{ \bigwedge_{i=1}^{n} c_i \approx d_i \rightarrow c \approx d \mid \begin{smallmatrix} f(c_1, \ldots, c_n) \approx c \in \mathsf{Def} \\ f(d_1, \ldots, d_n) \approx d \in \mathsf{Def} \end{smallmatrix} \}$.

As a consequence we obtain the following decidability transfer result.

Theorem 2 ([13,8]). *If the theory extension $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ satisfies condition (Loc^{Ψ}) then satisfiability of ground clauses G w.r.t. $\mathcal{T}_1$ is decidable provided $\mathcal{K}[\Psi_{\mathcal{K}}(G)]$ is finite and $\mathcal{K}_0 \cup G_0 \cup \Gamma_0 \cup \mathsf{Con}_0$ belongs to a decidable fragment of $\mathcal{T}_0$.*

Similar results hold for extended Ψ-local extensions (then $\mathcal{K}, \mathcal{K}_0$ and G_0 may contain arbitrary Π_0-sentences and G is a set of Σ_0-extended ground clauses).

The (Ψ-)locality of an extension can be recognized by proving embeddability of partial into total models assuming that the extension clauses are flat and linear [13,8,9]. If we can guarantee that the support of the total model which we obtain is the same as the support of the partial model we start with, then condition (ELoc^{Ψ}) is guaranteed. The locality proof also explains how to construct models of satisfiable ground (extended) clauses starting from models of their instances.

We present some theory extensions which are Ψ-local. In what follows we will denote by $\mathbb{R}$ both the set of real numbers and the theory of real numbers (more precisely the theory of real closed fields) and by $LI(\mathbb{R})$ linear arithmetic over $\mathbb{R}$.

Monotonicity, Boundedness for Monotone Functions. Any extension of a theory for which $\leq$ is a partial order with functions satisfying[4] $\mathsf{Mon}^{\sigma}(f)$ and/or $\mathsf{Bound}^{t}(f)$ is local (cf. e.g. [8]). The extensions satisfy condition (ELoc) if e.g. in $\mathcal{T}_0$ all finite and empty infima (or suprema) exist.

$\mathsf{Mon}^{\sigma}(f)$ $\quad \bigwedge_{i \in I} x_i \leq^{\sigma_i} y_i \wedge \bigwedge_{i \notin I} x_i \approx y_i \rightarrow f(x_1, .., x_n) \leq f(y_1, .., y_n)$

$\mathsf{Bound}^{s,t}(f)$ $\forall x_1, \ldots, x_n (s(x_1, \ldots, x_n) \leq f(x_1, \ldots, x_n) \leq t(x_1, \ldots, x_n))$

where s, t are new functions or $s(x_1, \ldots, x_n)$ and $t(x_1, \ldots, x_n)$ are Π_0-terms with variables among $x_1, \ldots, x_n$ and: (i) s, t have the same monotonicity as f in any model and (ii) $\forall x_1, \ldots, x_n \; s(x_1, \ldots, x_n) \leq t(x_1, \ldots, x_n)$.

Convexity/Concavity [14]. Let $\mathcal{T}_0$ be $\mathbb{R}$, the theory of real numbers (or the many-sorted combination of the theories of real numbers (sort real) and integers

[3] The function symbols in Σ_1 are separated from the other symbols by introducing, in a bottom-up manner, new constants c_t for subterms $t = f(c_1, \ldots, c_n)$ where $f \in \Sigma_1$ and c_i are constants, together with definitions $c_t = f(c_1, \ldots, c_n)$ (C is a set of constants containing all constants introduced by flattening and purification).

[4] For $i \in I$, $\sigma_i \in \{-, +\}$, and for $i \notin I$, $\sigma_i = 0$; $\leq^{+} = \leq, \leq^{-} = \geq$.

(sort int)). Let f be a unary function symbol with arity $\mathsf{real}\to\mathsf{real}$ (or resp. $\mathsf{int}\to\mathsf{real}$). Then $\mathcal{T}_0\subseteq\mathcal{T}_0\cup\mathsf{Conv}_f$ and $\mathcal{T}_0\subseteq\mathcal{T}_0\cup\mathsf{Conc}_f$ satisfy condition (ELoc) where:

$$\mathsf{Conv}(f)\ \ \forall x,y,z\left(x<z\leq y\to\frac{f(z)-f(x)}{z-x}\leq\frac{f(y)-f(x)}{y-x}\right).$$

and the concavity condition Conc_f is the convexity condition for $-f$.

Linear Combinations of Functions. Let $f_1,\ldots,f_n,f,g$ be unary function symbols. The extension $\mathbb{R}\subseteq\mathbb{R}\cup\mathsf{BS}$ satisfies condition (ELoc), where BS contains conjunctions of axioms of type $\forall t(a\leq\sum_{i=1}^{n}a_if_i(t)\leq b)$, $\forall t(g(t)\leq\sum_{i=1}^{n}a_if_i(t)\leq f(t))$, or $\forall t,t'(t<t'\to a\leq\sum_{i=1}^{n}a_i\frac{f_i(t')-f_i(t)}{t'-t}\leq b)$, where $a,b\in\mathbb{R}$ and (i) g is convex and f concave and (ii) either g and f satisfy the condition $\forall t(g(t)\leq f(t))$ or correspond to Π_0-terms and $\models_{\mathcal{T}_0}\forall t\ g(t)\leq f(t)$. Using arguments similar to those used in [14] it can be proved that if we additionally require the functions to be continuous locality is still preserved.

If I is an interval of the form $(-\infty,a]$, $[a,b]$ or $[a,\infty)$ then we can define versions of monotonicity/boundedness, convexity/concavity and boundedness axioms for linear combinations of functions and of their slopes relative to the interval I (then conditions (i) and (ii) for f and g are relative to the interval I).

Hybrid Automata. A hybrid automaton (abbreviated HA in what follows) [1] is a tuple $S=(X,Q,\mathsf{flow},\mathsf{Inv},\mathsf{Init},E,\mathsf{guard},\mathsf{jump})$ consisting of:

(1) A finite set $X=\{x_1,\ldots,x_n\}$ of real valued variables and a finite set Q of control modes, that together define the state space of the system;
(2) A family $\{\mathsf{flow}_q\mid q\in Q\}$ of predicates over the variables in $X\cup\dot{X}$ (where $\dot{X}=\{\dot{x}_1,\ldots,\dot{x}_n\}$, where $\dot{x}_i$ is the derivative of x_i) specifying the continuous dynamics in each control mode; a family $\{\mathsf{Inv}_q\mid q\in Q\}$ of predicates over the variables in X defining the invariant conditions for each control mode; and a family $\{\mathsf{Init}_q\mid q\in Q\}$ of predicates over the variables in X, defining the initial states for each control mode.
(3) A finite multiset E with elements in $Q\times Q$ (the control switches). Every $(q,q')\in E$ is a directed edge between q (source mode) and q' (target mode); a family of guards $\{\mathsf{guard}_e\mid e\in E\}$ (predicates over X); and a family of jump conditions $\{\mathsf{jump}_e\mid e\in E\}$ (predicates over $X\cup X'$, where $X'=\{x'_1,\ldots,x'_n\}$ is a copy of X consisting of "primed" variables).

A *state* of S is a pair (q,a) consisting of a control mode $q\in Q$ and a vector $a=(a_1,\ldots,a_n)$ that represents a value $a_i\in\mathbb{R}$ for each variable $x_i\in X$. A state (q,a) is *admissible* if Inv_q is true when each x_i is replaced by a_i. There are two types of *state change*: (i) A *jump* is an instantaneous transition that changes the control location and the values of data variables according to the jump conditions; (ii) In a *flow*, the state can change due to the evolution in a given control mode over an interval of time: the values of the data variables change continuously according to the flow rules of the current control location; all intermediate states are admissible. A *run* of S is a finite sequence $s_0s_1\ldots s_k$

of admissible states s_j such that (i) the first state s_0 is an initial state of S (the values of the variables satisfy Init_q for some $q \in Q$), (ii) each pair (s_j, s_{j+1}) is either a jump of S or the endpoints of a flow of S.
Notation. In what follows we use the following notation. If $x_1, \ldots, x_n \in X$ we denote the sequence $x_1, \ldots, x_n$ with $\overline{x}$, the sequence $\dot{x}_1, \ldots, \dot{x}_n$ with $\overline{\dot{x}}$, and the sequence of values $x_1(t), \ldots, x_n(t)$ of these variables at a time t with $\overline{x}(t)$.

Verification Problems. We are interested in invariant checking or bounded model checking *under given constraints on parameters*, and in *deriving constraints between parameters* which guarantee that a certain safety property is an invariant of the system or holds for paths of bounded length. We studied this problem in [15] and [3,4] but only for very simple types of linear hybrid systems. In this paper we continue this direction of research in two different ways:

Deductive Verification and Synthesis. We identify possibilities for generating conditions on parameters under which a certain formula (i) is an invariant or (ii) holds for paths of bounded length. For simplicity we will here focus on (i); (ii) can be solved in a similar way.

Criticality Functions. We give methods for generating *criticality functions* which ensure that a hybrid automaton satisfies certain safety conditions.

Example 1 (Running example). *We consider a temperature controller, modeled as a hybrid automaton with two modes: a heating mode (in which the environment of the object is heated) and a normal mode (heating is switched off). The control variable is x (the temperature of the object). We assume that the system has two parameters (which can be functional or not).*

- *The temperature of the heated environment (due to the heater): a constant h or (if it changes over time) a unary function h (input and output sort: real).*
- *Perturbation of the temperature of the environment due to external causes (e.g. external temperature), modeled using a constant f or (if it changes in time) a unary function f (input and output sort: real).*

Invariants and flows *in the two modes are described below ($k > 0$ is a constant which depends only on the surface of the object which is being heated):*

Mode 1 (Heating): *Invariant:* $T_a \leq x(t) \leq T_b$; *Flow:* $\frac{dx}{dt} = -k(x - (h + f))$,
Mode 2 (Normal): *Invariant:* $T_c \leq x(t) \leq T_d$; *Flow:* $\frac{dx}{dt} = -k(x - f)$.

Control Switches. *We have two control switches:*

e_{12}**:** *switch from Mode 1 to Mode 2 (if the temperature of the object becomes too high heating is switched off):* $\mathsf{guard}_{e_{12}}$: $x \geq T_b$; $\mathsf{jump}_{e_{12}}$: $(x' = x)$.
e_{21}**:** *switch from Mode 2 to Mode 1 (if the temperature of the object becomes too low heating is switched on):* $\mathsf{guard}_{e_{21}}$: $x \leq T_c$; $\mathsf{jump}_{e_{21}}$: $(x' = x)$.

Let $\mathsf{Safe} = T_m \leq x(t) \leq T_M$ *(a safety condition for the heater). Our goals are:*

(1) check that Safe *is an invariant (or that it holds on all runs of bounded length),*
(2) generate constraints which guarantee that Safe *is an invariant,*
(3) generate a criticality function which would provide a guarantee for safety.

In Sect. 3 we give methods for solving (1) and (2) for increasingly larger classes of parametric hybrid automata. We then solve (3) by using methods for model generation in local theory extensions (Sect. 4).

3 Deductive Verification and Synthesis

We present a general result which will be used for the verification of parametric linear hybrid automata (Sect. 3.1) and, in extended form also for more general classes of parametric hybrid automata, possibly interconnected (Sect. 3.2–3.3).

Let $\mathcal{T}_0$ be a Π_0-theory and let Σ_P be a set of parameters (function and constant symbols). Let Σ be a signature such that $\Sigma \cap (\Sigma_0 \cup \Sigma_P) = \emptyset$, containing functions which can change their values during flows and jumps. Let $\Sigma' = \{f' \mid f \in \Sigma\}$ be a disjoint copy of Σ representing the new functions after the updates. Let $\mathcal{KF}$ be a set of clauses over $\Pi_0 \cup \Sigma \cup \Sigma_P$ expressing the properties of the functions in $\Sigma \cup \Sigma_P$, and let $\mathsf{Update}(\overline{x}, \overline{x}', \overline{f}, \overline{f}')$ (containing for every x also x' and for every $f \in \Sigma$ also $f' \in \Sigma'$) be a set of clauses expressing the way the variables and the functions in Σ are changed during updates.

Let Ψ be a set of clauses in the signature $\Pi_0 \cup \Sigma_P \cup \Sigma$ which can contain (implicitly universally quantified) variables as arguments of the functions in $\Sigma_P \cup \Sigma$.

Example 2. *In Example 1, $\mathcal{T}_0$ is the theory of real numbers. The parameters in Σ_P are $T_m, T_M, T_a, T_b, T_c, T_d$ (constants) and h, f (functions). The set of functions which change their values is $\Sigma = \{x\}$; $\Sigma' = \{x'\}$. An example of set of axioms describing the properties of parameters is: $\mathcal{KF} = \{T_m \leq T_M, T_a \leq T_c \leq T_b \leq T_d, k \geq 0, \forall t(0 \leq h(t))\}$. We have two types of updates: updates due to flows (in which x is the value of the variable x at the beginning of the flow and x' the value at the end of the flow) and updates due to jumps (described by the jump conditions). More complex examples are given in Sect. 3.1–3.3.*

Theorem 3 ([15]). *Let Γ_0 be a set of constraints on the parameters. Assume that $\Gamma_0, \Psi, \mathcal{KF}$ and* Update *are sets of clauses which define the following chain of theory extensions satisfying condition* ELoc*: $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \Gamma_0 \subseteq \mathcal{T}_0 \cup \Gamma_0 \cup \Psi \subseteq \mathcal{T}_0 \cup \Gamma_0 \cup \Psi \cup \mathcal{KF} \subseteq \mathcal{T}_0 \cup \Gamma_0 \cup \Psi \cup \mathcal{KF} \cup$ Update. Assume that either (i) ground satisfiability in $\mathcal{T}_0$ is decidable and all variables in these theory extensions occur below extension functions, or (ii) satisfiability of $\exists\forall$ formulae is decidable in $\mathcal{T}_0$. Then we can decide whether Ψ is an invariant (using repeatedly Theorem 1).*

If $\mathcal{T}_0$ has quantifier elimination then we can construct a constraint on the parameters Γ which guarantees that Ψ is an invariant:

Theorem 4. *Assume that the theory $\mathcal{T}_0$ has quantifier elimination. We can construct a universally quantified formula $\forall \overline{x} \Gamma(\overline{x})$ (containing also some of the parameters) such that for every structure $\mathcal{A}$ with signature $\Pi_0 \cup \Sigma \cup \Sigma' \cup \Sigma_P$ which is a model of $\mathcal{T}_0$, if $\mathcal{A} \models \forall \overline{x} \Gamma(\overline{x})$ then Ψ is an invariant w.r.t. interpretation $\mathcal{A}$.*

Proof: We use instantiation, renaming (as in Thm. 1) and quantifier elimination in $\mathcal{T}_0$ to construct a formula Γ such that if Ψ is not invariant w.r.t. $\mathcal{A}$, i.e.

$\mathcal{A} \models \mathcal{T}_0 \cup \mathcal{KF} \cup \exists \overline{x}, \overline{x}'(\Psi(\overline{x}.\overline{f}) \wedge \mathsf{Update}(\overline{x}, \overline{x}', \overline{f}, \overline{f}') \wedge \neg\Psi(\overline{x}', \overline{f}'))$ then $\mathcal{A}$ is a model of $\exists x \neg\Gamma(x)$, so, if $\mathcal{A} \models \forall \overline{y}\Gamma(\overline{y})$ then Ψ is an invariant w.r.t. $\mathcal{A}$. □

In what follows we present applications of this result to increasingly more complex classes of hybrid automata.

3.1 Parametric Linear Hybrid Automata

A hybrid automaton S is a linear hybrid automaton (LHA) if it satisfies the following two requirements [1]:

1. Linearity: For every control mode $q \in Q$, the flow condition flow_q, the invariant condition Inv_q, and the initial condition Init_q are convex linear predicates[5]. For every control switch $e = (q, q') \in E$, the jump condition jump_e and the guard guard_e are convex linear predicates. In addition, as in [3,4], we assume that the flow conditions flow_q are conjunctions of *non-strict* linear inequalities.

2. Flow independence: For every control mode $q \in Q$, the flow condition flow_q is a predicate over the variables in $\dot{X}$ only (and does not contain any variables from X). This requirement ensures that the possible flows are independent from the values of the variables, and only depend on the control mode.

We also consider *parametric* linear hybrid automata (PLHA), defined as linear hybrid automata for which a set $\Sigma_P = P_c \cup P_f$ of parameters is specified (consisting of parametric constants P_c and parametric functions P_f) with the difference that for every control mode $q \in Q$ and every mode switch e:

(1) the linear constraints in the invariant conditions Inv_q, initial conditions Init_q, and guard conditions guard_e are of the form: $g \leq \sum_{i=1}^{n} a_i x_i \leq f$,
(2) the inequalities in the flow conditions flow_q are of the form: $\sum_{i=1}^{n} b_i \dot{x}_i \leq b$,
(3) the linear constraints in jump_e are of the form $\sum_{i=1}^{n} b_i x_i + c_i x_i' \leq d$,

(possibly relative to an interval I) where the coefficients a_i, b_i, c_i and the bounds b, d are either numerical constants or parametric constants in P_c; and g and f are (i) constants or parametric constants in P_c, or (ii) parameteric functions in P_f satisfying the convexity (for g) resp. concavity condition (for f), or concrete functions with these convexity/concavity properties such that $\forall t(g(t) \leq f(t))$. The flow independence conditions hold as in the case of linear hybrid automata.

Note: In the definition of PLHA we allow a general form of parametricity, in which the bounds in state invariants, guards and jump conditions can be expressed using functions with certain properties. Such parametric descriptions of bounds are useful for instance in situations in which we want to verify systems which have non-linear behavior and use a parametric approximation for them.

Example 3. *Consider the hybrid automaton S presented in Example 1. If in the heating mode the invariant is $T_a \leq x(t) \leq T_b$ and the flow is $\frac{dx}{dt} = -k(x-(h+f))$, where f, h are constants, then we can approximate the flow by the linear flow:*

$$-k(T_b - (h+f)) \leq \dot{x} \leq -k(T_a - (h+f)). \quad (1)$$

[5] An atomic linear predicate is a strict or non-strict linear inequality. A convex linear predicate is a finite conjunction of linear inequalities.

We can obtain similar bounds for $\dot{x}$ also for mode 2. Thus, we can approximate S using a linear hybrid automaton S'. If we can guarantee safety in S', then safety is preserved for all possible runs which satisfy the flow conditions of S', in particular also for all runs of S, so S is safe.

We give methods of deciding whether a formula Ψ is an invariant and of deriving conditions that guarantee that a PLHA S has a given safety property. To use Thm. 4, we analyze the possible updates in PLHA by jumps and flows.

Jumps. A jump update can be expressed by the linear inequality
$\mathsf{Jump}_e(\overline{x}, \overline{x}') = \mathsf{guard}_e(\overline{x}) \wedge \mathsf{jump}_e(\overline{x}, \overline{x}')$.

Flows. Assume that $\mathsf{flow}_q(t) = \bigwedge_{j=1}^{n_q}(\sum_{i=1}^{n} c_{ij}^q \dot{x}_i(t) \leq_j c_j^q)$. We alternatively axiomatize flows in mode q in the time interval $[t_0, t_1]$ (where $0 \leq t_0 \leq t_1$) as follows:

$$\mathsf{Flow}_q(t_0, t_1) = \forall t(t_0 \leq t \leq t_1 \rightarrow \mathsf{Inv}_q(\overline{x}(t))) \wedge \forall t, t'(t_0 \leq t < t' \leq t_1 \rightarrow \underline{\mathsf{flow}}_q(t, t'))$$

where: $\underline{\mathsf{flow}}_q(t, t') = \bigwedge_{j=1}^{n_q}(\sum_{i=1}^{n} c_{ij}^q(x_i(t') - x_i(t)) \leq_j c_j^q(t' - t))$.

In [3,4] we showed that for LHA no precision is lost with this axiomatization. We can, in fact simplify the axiomatization of flows further by suitably instantiating the universal quantifiers in Flow_q and obtain:

Theorem 5 ([4]). *The following are equivalent for any LHA:*

(1) Ψ is an invariant of the automaton.
(2) For every $q \in Q$ and $e = (q, q') \in E$, the formulae $F_{\mathsf{init}}(q)$, $F'_{\mathsf{flow}}(q)$ and $F_{\mathsf{jump}}(e)$ are unsatisfiable, where:

$F_{\mathsf{Init}}(q)$ $\quad \mathsf{Init}_q(\overline{x}(t_0)) \wedge \neg\Psi(\overline{x}(t_0))$
$F'_{\mathsf{flow}}(q)$ $\quad \mathsf{Inv}_q(\overline{x}(t_0)) \wedge \Psi(\overline{x}(t_0)) \wedge \underline{\mathsf{flow}}_q(t_0, t) \wedge \mathsf{Inv}_q(\overline{x}(t)) \wedge \neg\Psi(\overline{x}(t)) \wedge t \geq t_0$
$F_{\mathsf{jump}}(e)$ $\quad \Psi(\overline{x}(t)) \wedge \mathsf{Jump}_e(\overline{x}(t), \overline{x}'(0)) \wedge \mathsf{Inv}_{q'}(\overline{x}'(0)) \wedge \neg\Psi(\overline{x}'(0))$.

Let S be a parametric LHA with parameters $\Sigma_P = P_c \cup P_f$. Assume that the properties of the parameters are expressed as $\Gamma_0 \wedge \Gamma_f$, where Γ_0 is a conjunction of linear inequalities representing the relationships between parameters in P_c and Γ_f is a set of (universally quantified) clauses expressing the properties of the functional parameters (in P_f) – containing the convexity/concavity conditions for the bounding functional parameters. Let Ψ be a property expressed as convex linear predicate over X, possibly containing parameters (constants as coefficients; either constants or functions as bounds in the linear inequalities). In [4] we showed that checking whether Ψ is an invariant is decidable in PTIME if bounds are not functional and in EXPTIME otherwise.[6]

Since the theory of reals allows quantifier elimination, the following result is a direct consequence of Thm. 4.

[6] The proof in [4] is direct; this result can also be proved using the fact that extensions with axioms expressing boundedness conditions on linear combinations of functions and of their slopes define local theories and therefore (i) updates by flows as well as updates by jumps define local theory extensions for any extension of the reals; (ii) any convex predicate Ψ defines a local extension of the theory of reals.

Theorem 6. *Let S be a PLHA and Ψ be a property expressed as a convex linear predicate over X, possibly containing parameters. We can effectively derive a set Γ of (universally quantified) constraints on the parameters such that whenever Γ holds in an interpretation $\mathcal{A}$, Ψ is an invariant w.r.t. $\mathcal{A}$.*

This method for constraint synthesis can in particular be used for:

1. Invariant generation: Let S be a fixed (non-parametric) LHA. We consider invariant "templates" Ψ expressed by linear inequalities with parametric bounds and coefficients and determine constraints on these parameters which ensure that Ψ is an invariant. By finding values of the parameters satisfying these constraints we can generate concrete invariants.

2. Generation of control conditions: Assume that Ψ is fixed (non-parametric), but that the mode invariants, the flow conditions, the guards, and the jumps are represented parametrically (as conjunctions of a bounded number of linear inequalities). We can determine constraints on the parameters which ensure that Ψ is an invariant. By finding values of the parameters satisfying these constraints we can determine control conditions which guarantee that Ψ is invariant.

Example 4. *Consider a variant of the HA in Example 1 in which T_a, T_b are functional parameters, Inv_1 is $\forall t(T_a(t) \leq x(t) \leq T_b(t))$, and the flow in mode 1 is described by: $-k(x_b - g) \leq \dot{x} \leq -k(x_a - g)$ (for simplicity we abbreviated $h + f$ by g). Let $\Psi = (T_m \leq x(t) \leq T_M)$. Assume that the properties of the parameters are axiomatized by $\mathcal{KF}$: $\{\forall t(x_a \leq T_a(t)), \forall t(T_b(t) \leq x_b),\ x_a < x_b,\ T_m < T_M\}$. We derive a condition Γ which guarantees that Ψ is preserved under flows in mode 1 using Thm. 4 as follows. Consider the formula:*

$$\begin{array}{l}\exists t_0, t_1\ (T_m \leq x(t_0) \leq T_M)\ \wedge\ (T_a(t_0) \leq x(t_0) \leq T_b(t_0))\ \wedge \\ \qquad (x(t_1) \leq x(t_0) - k(t_1 - t_0)(x_a - g) \wedge x(t_0) - k(t_1 - t_0)(x_b - g) \leq x(t_1)) \\ \qquad (T_a(t_1) \leq x(t_1) \leq T_b(t_1))\ \wedge\ (x(t_1) < T_m\ \ \vee\ \ T_M < x(t_1)).\end{array}$$

After purification we obtain:

$$\begin{array}{l}\exists t_0, t_1, T_{a0}, T_{b0}, x_0, x_1 (T_m \leq x_0 \leq T_M)\ \wedge\ (T_{a0} \leq x_0 \leq T_{b0})\ \wedge \\ \qquad (x_1 \leq x_0 - k(t_1 - t_0)(x_a - g) \wedge x_0 - k(t_1 - t_0)(x_b - g) \leq x_1)\ \wedge \\ \qquad (T_{a1} \leq x_1 \leq T_{b1})\ \wedge\ (x_1 < T_m\ \ \vee\ \ T_M < x_1).\end{array}$$

After eliminating the quantifiers $\exists x_0, x_1$ (assuming $\mathcal{KF}$ holds and $T_m \leq T_M$) and then replacing back T_{a0}, T_{a1} and T_{b0}, T_{b1} we obtain:

$$\begin{array}{l}\exists t_0, t_1\ (T_a(t_0) \leq T_M \wedge T_m \leq T_b(t_0) \wedge \\ \qquad T_m \leq T_b(t_1) + k(t_1 - t_0)(x_b - g) \wedge T_a(t_0) \leq T_b(t_1) + k(t_1 - t_0)(x_b - g) \wedge \\ \qquad T_a(t_1) + k(t_1 - t_0)(x_a - g) \leq T_M \wedge T_a(t_1) + k(t_1 - t_0)(x_a - g) \leq T_b(t_0) \wedge \\ \qquad [((k(t_1 - t_0)(x_b - g) > 0) \wedge T_a(t_1) < T_m \wedge T_a(t_0) < T_m + k(t_1 - t_0)(x_b - g)) \vee \\ \qquad\ ((k(t_1 - t_0)(x_a - g) < 0) \wedge T_b(t_1) > T_M \wedge T_M + k(t_1 - t_0)(x_a - g) < T_b(t_0))]).\end{array}$$

The condition Γ_1 which ensures that Ψ is an invariant under flows in mode 1 is:

$$\begin{array}{l}\forall t_0, t_1\ ((T_a(t_0) \leq T_M \wedge T_m \leq T_b(t_0) \wedge \\ \qquad T_m \leq T_b(t_1) + k(t_1 - t_0)(x_b - g) \wedge T_a(t_0) \leq T_b(t_1) + k(t_1 - t_0)(x_b - g) \wedge \\ \qquad T_a(t_1) + k(t_1 - t_0)(x_a - g) \leq T_M \wedge T_a(t_1) + k(t_1 - t_0)(x_a - g) \leq T_b(t_0) \\ \quad \rightarrow [((k(t_1 - t_0)(x_b - g) \leq 0) \vee T_a(t_1) \geq T_m \vee T_a(t_0) \geq T_m + k(t_1 - t_0)(x_b - g)) \wedge \\ \qquad ((k(t_1 - t_0)(x_a - g) \geq 0) \vee T_b(t_1) \leq T_M \vee T_M + k(t_1 - t_0)(x_a - g) \geq T_b(t_0))]).\end{array}$$

Invariant generation. *Assume that the control of the LHA (i.e. the functions T_a, T_b and their bounds x_a, x_b) is fixed, e.g. $k = 1$, T_a is the constant function $x_a = 15$ and T_b is the constant function $x_b = 20$, $h = 25$ and $f = 10$. We can use the constraint in Γ_1 to determine for which values of the constants T_m and T_M Ψ is an invariant under flows in mode 1 (we can check that it is, e.g., for $T_m = 15$ and $T_M = 20$). The generation of control conditions for guaranteeing that a (non-parametric) Ψ is invariant is similar.*

3.2 Parametric Hybrid Automata

We now extend the methods developed above for parametric LHA to more general HA. For the sake of simplicity, we only consider (parametric) hybrid automata S with one continuous variable x. (The case when we have several variables is similar, but the presentation is more complicated.) Let $\Sigma = \{x\}$ and $\Sigma' = \{x'\}$. Assume that mode invariants, initial states, guards and jump conditions are expressed as sets of clauses in an extension of the theory of real numbers with additional functions in a set $\Sigma_1 = \Sigma \cup \Sigma_P$, where Σ_P is a set of parameter names (both functions and constants). We study the problem of deriving constraints on parameters which guarantee that a certain formula is an invariant. We therefore analyze the possible updates by jumps and flows.

Jumps. Assume that the guards and the jump conditions are given by formulae in a certain extension of the theory of real numbers. A jump update can be expressed by the formula: $\mathsf{Jump}_e(x, x') = \mathsf{guard}_e(x) \wedge \mathsf{jump}_e(x, x')$.

Flows. In the case of parametric hybrid automata, the flows are described by differential equations. As we restrict to the case of one continuous variable x, we assume that in mode q the flow is described by: $\frac{dx}{dt}(t) = f_q(x(t))$. Thus:

$$\mathsf{Flow}_q(t_0, t_1) = \forall t(t_0 \leq t \leq t_1 \rightarrow \mathsf{Inv}_q(x(t))) \wedge \forall t(t_0 \leq t \leq t_1 \rightarrow \tfrac{dx}{dt}(t) = f_q(x(t))).$$

Let Ψ be a set of clauses in the signature $\Pi_0 \cup \Sigma_P \cup \Sigma$ which can contain (implicitly universally quantified) variables as arguments of the functions in $\Sigma_P \cup \Sigma$.

Theorem 7. *(1) For every jump e we can construct a universally quantified formula $\forall \overline{x} \Gamma_e(\overline{x})$ (containing also some of the parameters) such that for every structure $\mathcal{A}$ with signature $\Pi_0 \cup \Sigma \cup \Sigma' \cup \Sigma_P$ if $\mathcal{A}$ is a model of $\mathcal{T}_0$ and of Γ_e then Ψ is an invariant under the jump e (in interpretation $\mathcal{A}$).*

(2) For every flow in a mode q we can construct a universally quantified formula $\forall \overline{x} \Gamma_q(\overline{x})$ (containing also some of the parameters) such that for every structure $\mathcal{A}$ with signature $\Pi_0 \cup \Sigma \cup \Sigma' \cup \Sigma_P$ if $\mathcal{A}$ is a model of $\mathcal{T}_0$ and of Γ_q then Ψ is an invariant under flows in q (in interpretation $\mathcal{A}$).

Proof: (1) follows from Thm. 4 for the case of jump updates. (2) Assume that $\mathcal{A}$ is a model in which Ψ is not invariant under flows in mode q. Then there exist time points $t_0 < t_1$ such that (i) $\mathcal{A} \models \forall t(t_0 \leq t \leq t_1 \rightarrow \mathsf{Inv}_q(x(t)))$ and (ii) the interpretation in $\mathcal{A}$ of the function x, $x_A : \mathbb{R} \rightarrow \mathbb{R}$ is differentiable and has the property that $\forall t(t_0 \leq t \leq t_1 \rightarrow \frac{dx}{dt}(t) = f_q(x(t)))$. Then, by the mean value theorem:

$$\mathcal{A} \models \Psi(t_0) \wedge \forall t(t_0 \leq t \leq t_1 \rightarrow \mathsf{Inv}_q(x(t))) \wedge$$
$$\forall t, t'(t_0 \leq t < t' \leq t_1 \rightarrow \exists c(t \leq c \leq t' \wedge \tfrac{x(t') - x(t)}{t' - t} = f_q(x(c)))) \wedge \neg\Psi(x(t_1)).$$

Therefore, $\mathcal{A}$ is a model of any set of instances of the formula above, in particular of those instances in which the universally quantified variables are instantiated with the constants $\{t_0, t_1\}$ (we can take more or fewer instances, depending on how strong we want the condition Γ_q to be). For every choice of instances for the pair of variables t, t' in the flow description above we need to replace the existentially quantified variable c with a new constant. We can now continue as in the proof of Thm. 4 and obtain the formula Γ_q. □

Example 5. *Consider a variant of the HA in Example 1 in which f and h are unary functions, and the invariants and flow in the two modes are described by:*

Mode 1 (Heating): *Invariant:* $\mathsf{Inv}_1(x(t)) := x(t) \leq T_M$
Flow: $\frac{dx}{dt}(t) = -k(x(t) - (h(t) + f(t)))$
Mode 2 (Normal): *Invariant:* $\mathsf{Inv}_1(x(t)) := T_m \leq x(t)$
Flow: $\frac{dx}{dt}(t) = -k(x(t) - f(t))$

Let $\Psi(t) = T_m \leq x(t) \leq T_M$. We derive constraints which guarantee that this Ψ is invariant in mode 2. We proceed as in the proof of Theorem 7: If $\mathcal{A}$ is an interpretation in which Ψ is not invariant then there exist $t_0, t_1 \in \mathbb{R}$ such that:

(a) $t_0 < t_1 \wedge \forall t(t_0 \leq t \leq t_1 \rightarrow T_m \leq x(t))$
(b) $\forall t', t''(t_0 \leq t' < t'' \leq t_1 \rightarrow \exists y(t \leq y \leq t' \wedge \frac{x(t')-x(t)}{t'-t} = -k(x(y) - f(y))))$
(c) $(T_m \leq x(t_0) \wedge x(t_0) \leq T_M) \quad \wedge \quad (x(t_1) > T_M \vee x(t_1) < T_m)$

This also holds if we instantiate t with t_0, and t_1; t' with t_0; and t'' with t_1, so:

$\mathcal{A} \models (T_m \leq x(t_0)) \wedge (T_m \leq x(t_1)) \wedge \exists y(t_0 \leq y \leq t_1 \wedge \frac{x(t_1)-x(t_0)}{t_1-t_0} = -k(x(y) - f(y))) \wedge$
$(T_m \leq x(t_0) \wedge x(t_0) \leq T_M) \wedge (x(t_1) > T_M \vee x(t_1) < T_m)$

We know that there exists a value for y, say $a \in \mathbb{R}$ for which the formula holds. We introduce a new constant c; let $\mathcal{A}^c$ be the expansion of $\mathcal{A}$ in which the interpretation of c is $c_A := a$. Then:

$\mathcal{A}^c \models (T_m \leq x(t_0)) \ \wedge \ (T_m \leq x(t_1)) \ \wedge \ \frac{x(t_1)-x(t_0)}{t_1-t_0} = -k(x(c) - f(c)) \wedge$
$(T_m \leq x(t_0) \wedge x(t_0) \leq T_M) \quad \wedge \quad (x(t_1) > T_M \vee x(t_1) < T_m)$

We purify the formula introducing the abbreviations: $d_0 = x(t_0)$, $d_1 = x(t_1)$ and $d = x(c)$ and $d_f = f(c)$. We obtain the following set of constraints:

$\exists c, d, d', t_0, t_1, c_0, c_1 \ [T_m \leq c_0 \wedge T_m \leq c_1 \wedge \frac{c_1-c_0}{t_1-t_0} = -k(d - d_f) \wedge$
$(T_m \leq c_0 \wedge c_0 \leq T_M) \quad \wedge \quad (c_1 > T_M \vee c_1 < T_m)]$

We eliminate the variables t_0, t_1, c_0, and c_1 (we used REDLOG [5]) under the assumption that $t_0 < t_1$ and $t_0 \leq c \leq t_1$ and obtain the equivalent formula:

$$\exists c, d, d'(-k(d - d_f) > 0 \quad \wedge \quad T_M > T_m)$$

We now replace again the constants d and d_f with the terms they represent and obtain: $\exists c(-k(x(c) - f(c)) > 0 \ \wedge \ T_M > T_m)$. We can conclude that if $T_M > T_m$ then Ψ is invariant under all flows in mode 2 if $\forall c \ \ k*(f(c) - x(c)) \leq 0$ (i.e. if the system does not heat in this state because of the external temperature). Of course this is only a sufficient condition.

3.3 Interconnected Families of Hybrid Automata

We can also consider systems of interconnected parametric hybrid automata $\{S_1, \ldots, S_n\}$ with a parametric number of components under the assumptions:

(1) The invariants, guards, jump and flow conditions of $S_1, \ldots, S_n$ can all be expressed similarly (and can be written globally, using indices);
(2) The relationships between the hybrid automata are uniform (and can again be expressed globally using indices);
(3) The topology of the system can be represented using data structures (e.g. arrays, lists, trees).

A general formalization of such situations is out of the scope of this paper. We here present the ideas on an example.

Example 6. *Consider a family of n water tanks with a uniform description, modeled by hybrid automata $S_1, \ldots, S_n$. Assume that every S_i has one continuous variable L_i (representing the water level in S_i), and that the input and output in mode q are described by parameters in_i and out_i. Every S_i has one mode in which the water level evolves according to rule $\dot{L}_i = \mathsf{in}_i - \mathsf{out}_i$. We write $L(i,t), \mathsf{in}(i)$ and $\mathsf{out}(i)$ instead of $L_i(t), \mathsf{in}_i$ and resp. out_i.*
Assume that the water tanks are interconnected in such a way that the input of system S_{i+1} is the output of system S_i. A global constraint describing the communication of the systems is therefore:

$$\forall i(2 \leq i \leq n-1 \rightarrow (\mathsf{in}(i) = \mathsf{out}(i-1)) \quad \wedge \quad \mathsf{in}(1) = \mathsf{in}.$$

An example of a "global" update describing the evolution of the systems S_i during a flow in interval $[t_0, t_1]$:

$$\forall i(L(i,t_1) = L(i,t_0) + (\mathsf{in}(i) - \mathsf{out}(i))(t_1 - t_0)).$$

Let $\Psi(t) = \forall i(L(i,t) \leq L_{\mathsf{overflow}})$. Assume that $\forall i(\mathsf{in}(i) \geq 0 \wedge \mathsf{out}(i) \geq 0)$. We generate a formula which guarantees that Ψ is an invariant as in the proof of Thm. 4. We start with the following formula (for simplicity of presentation we already replaced $\mathsf{in}(i)$ with $\mathsf{out}(i-1)$):

$$\begin{array}{l}\exists t_0, t_1\ t_0 < t_1 \wedge (\forall i(L(i,t_0) \leq L_{\mathsf{overflow}}) \wedge \exists j(L(j,t_1) > L_{\mathsf{overflow}})) \wedge \\ \quad \forall i((i = 1 \wedge L(1,t_1) = L(1,t_0) + (\mathsf{in} - \mathsf{out}(1))(t_1 - t_0)) \vee \\ \quad\quad (i > 1 \wedge L(i,t_1) = L(i,t_0) + (\mathsf{out}(i-1) - \mathsf{out}(1))(t_1 - t_0))).\end{array}$$

We skolemize (replacing j with the constant i_0) and instantiate all universally quantified variables i in the formula with i_0. After replacing $T(i_0, t_j)$ with c_j, $\mathsf{out}(i_0 - 1)$ with d_1, and $\mathsf{out}(i_0)$ with d_2 we obtain:

$$\begin{array}{l}\exists t_0, t_1 \exists c_0, c_1 [t_0 < t_1 \wedge (c_0 \leq L_{\mathsf{overflow}}) \wedge c_1 > L_{\mathsf{overflow}} \wedge \\ ((i_0 = 1 \wedge c_1 = c_0 + (\mathsf{in} - d_2)(t_1 - t_0)) \vee (i_0 > 1 \wedge c_1 = c_0 + (d_1 - d_2)(t_1 - t_0)))].\end{array}$$

We eliminate c_1 and c_0 using quantifier elimination and obtain:

$$\exists t_0, t_1 [t_0 < t_1 \wedge ((i_0 = 1 \wedge -(\mathsf{in} - d_2)(t_1 - t_0) < 0) \vee (i_0 > 1 \wedge -(d_1 - d_2)(t_1 - t_0) < 0))].$$

This is equivalent (after eliminating also t_0, t_1) with:

$$((i_0 = 1 \wedge (\mathsf{in} - d_2) > 0) \vee (i_0 > 1 \wedge (d_1 - d_2) > 0)).$$

We replace d_1, d_2 back and reintroduce the existential quantifier for i_0:

$$\exists i_0((i_0 = 1 \wedge (\mathsf{in} - \mathsf{out}(i_0)) > 0) \vee (i_0 > 1 \wedge (\mathsf{out}(i_0 - 1) - \mathsf{out}(i_0)) > 0)).$$

The negation is $\forall i((i{=}1 \rightarrow (\mathsf{in}{-}\mathsf{out}(i_0)){\leq}0) \wedge (i{>}1 \rightarrow (\mathsf{out}(i{-}1){-}\mathsf{out}(i)){\leq}0))$. *This condition guarantees that* Ψ *is an invariant for the family of systems.*

Similar results can be obtained if updates are caused by changes in topology (insertion or deletion of water tanks in the system).

4 Generating Criticality Functions

We study possibilities of proving safefy properties for hybrid automata by generating so-called criticality functions, i.e. functions which measure the "distance" (in a certain sense) from the current state and a safe state and are guaranteed to decrease during the evolution of the system. If such criticality functions can be constructed then the system can be proved to be safe. For keeping the notation and the proofs simple, we consider hybrid automata with only one continuous variable x. All definitions below are formulated for this special case.

Definition 8. *Let* S *be a hybrid automaton with a finite set* Q *of modes s.t.: (i) the dynamics in mode* q *is given by* $\frac{dx}{dt}(t) = f_q(x(t))$*, and (ii) the invariant of mode* q *is* Inv_q*. A family of* criticality functions *for this system is a family of maps* $c_q : \mathbb{R} \rightarrow \mathbb{R}$*,* $q \in Q$ *with the following properties:*

(1) $\forall x, c_q(x) \geq 0$*;*

(2) for all $q \in Q$*, whenever* $\mathsf{Inv}_q(x)$ *holds, we have* $\frac{dc_q}{dx}\frac{dx}{dt} \leq 0$*;*

(3) for all mode switches $e = (q_1, q_2) \in E$ *and all* x, x' *with* $\mathsf{guard}_e(x) \wedge \mathsf{jump}_e(x, x')$ *we have* $c_{q_2}(x') \leq c_{q_1}(x)$*.*

Let $\mathsf{Safe}(x)$ *be a safety property expressed by a formula with free variable* x*, containing* x *but not* $\dot{x}$*. Then the family* $\{c_q\}_{q \in Q}$ *is a family of criticality functions relative to* Safe *if there exists* $k > 0$ *such that for all* $q \in Q$*:*

(4) For all $x \in \mathbb{R}$*,* $(\mathsf{Inv}_q(x) \rightarrow (\mathsf{Safe}(x) \leftrightarrow c_q(x) \leq k))$*.*

By (2) and (3), c_q will never increase throughout any run of the system. By (4), it is impossible for a run beginning in a state satisfying Safe to end in a state which does not satisfy Safe, as this would require an increase of $c_q(x)$.

Techniques used for computing Lyapunov functions can also be used for computing criticality functions (linear matrix inequalities, convex feasibility problem). In [12], in order to compute criticality functions the hybrid system is decomposed as follows: strongly connected components/cycles are detected; predicates at border nodes are used in order to obtain smaller optimization problems.

Our goal is to exploit simpler properties of the functions describing the evolution of state vectors at each mode to compute a family of criticality functions. We will focus on monotonicity properties.

Assumption 1. In what follows we assume that for every mode $q \in Q$ we have: (1) Inv_q defines a convex set and (2) if the flow condition in q is $\frac{dx}{xt}(t) = f_q(x(t))$ then the function f_q is continuous and the set $\{x \in \mathbb{R} \mid f_q(x) = 0\}$ is finite.

Remark 9. *Let* S *be a hybrid system. Assume that the flow in mode* q *is given by* $\frac{dx}{dt}(t) = f_q(x(t))$*. Let* $I \subseteq \{x \mid \mathsf{Inv}_q(x)\}$*. If* $f_q(x(t)) \geq 0$ *for all* $t \in I$ *then* x *is increasing on* I *and if* $f_q(x(t)) \leq 0$ *for all* $t \in I$ *then* x *is decreasing on* I*.*

Lemma 10. *Let S be a HA satisfying Assumption 1, with real valued variable x and set of modes Q such that for every $q \in Q$ the flow in mode q is given by $\frac{dx}{dt}(t) = f_q(x(t))$. We can construct a HA S' with the same variable x and with set of modes Q' with the property that: (1) in every mode $q \in Q'$ x is monotone (either decreasing or increasing), and (2) for every run in S from state s to state s' there exists a run in S' from s to s' and vice versa.*

Proof: We construct S' as follows: For every mode q we analyze the sign changes of the function f_q. Let $x_1, y_1, \ldots, x_n, y_n \in \mathbb{R}$ with $x_1 < y_1 \ldots x_n < y_n$ be all the points at which f_q changes its sign. (For instance we can have $f_q(x) \leq 0$ for all $x \in (-\infty, x_1]$, $f_q(x) \geq 0$ for all $x \in [x_i, y_i]$; $f_q(x) \leq 0$ for all $x \in [y_i, x_{i+1}]$; and $f_q(x) \leq 0$ for all $x \in [y_n, \infty)$.) Let $\mathsf{Pos}_{f_q}(x) = \{x \in \mathbb{R} \mid f_q(x) \geq 0\}$ and $\mathsf{Neg}_{f_q}(x) = \{x \in \mathbb{R} \mid f_q(x) \leq 0\}$. Clearly, Pos_{f_q} is a union of intervals in a set I^q_{pos} and Neg_{f_q} is a union of intervals in a set I^q_{neg}. We split mode q accordingly in new modes $\{q_{i,I} \mid I \in I^q_{\mathsf{pos}}\}$ and $\{q_{d,J} \mid J \in I^q_{\mathsf{neg}}\}$ with invariants (in S'): $\mathsf{Inv}_{q_{u,I}} = \mathsf{Inv}_q \wedge C_I$ (C_I is the constraint characterizing interval I). For every $u \in \{i, d\}$ and every I, the flow in mode $q_{u,I}$ (in S') is the same as the flow in q in S. We have the same jumps from $q_{u,I}$ to $q'_{v,J}$ as between q and q' (provided the guards are compatible with C_I). In addition we can define in a natural way back and forth jumps between modes $q_{i,I}$ and $q_{d,J}$ for adjacent intervals I, J. It is easy to check that there exists a run in S from state s to state s' iff there exists a run in S' from s to s'. □

In the modes $q_{i,I}$, x is increasing as a function of time, so every criticality function is decreasing (as a function of x). Similarly every criticality function is increasing (as a function of x) in the modes $q_{d,J}$ where x is decreasing.

Theorem 11. *Let S be a HA satisfying Assumption 1, and let S' be the HA obtained from S as in Lemma 10. Let $\mathsf{Safe}(x)$ be a set of clauses with free variable x describing a safety condition and let $\mathcal{K}$ be the conjunction of the following axioms (where q ranges over the set Q' of nodes in S'):*

(1) $\forall x(c_q(x) \geq 0)$,
(2) axioms expressing that c_q is increasing in all modes of S' in which x is decreasing and decreasing in all modes of S' in which x is increasing,
(3) conditions ensuring that for every control switch (q_1, q_2), $c_{q_1}(x) \geq c_{q_2}(x')$,
(4) the condition that $\forall x(\mathsf{Inv}_q(x) \rightarrow (c_q(x) \leq k \leftrightarrow \mathsf{Safe}(x)))$.

We can decide whether $\mathcal{K}$ is satisfiable. Every model of $\mathcal{K}$ yields a family of criticality functions for S' and Safe *and thus also guarantees safety of S.*

Proof: It is easy to check that $\mathcal{K}$ defines a local theory extension, so it is satisfiable iff the set of instances in which the variables are instantiated with constants occurring in $\mathcal{K}$ is satisfiable. If $\mathcal{K}$ is satisfiable we can use methods for model generation for obtaining a family of functions with the desired properties. □

Example 7. *Consider a variant of the hybrid automaton from Example 1 where $\mathsf{Inv}_1 = x \leq T_M$ and $\mathsf{Inv}_2 = x \geq T_m$ and the functions describing the flow are $f_1(x) = -k(x - (h + f))$ for mode 1 and $f_2(x) = -k(x - f)$ for mode 2. Clearly,*

if $x \geq (h+f)$ then $f_1(x) \leq 0$ and if $x \leq (h+f)$ then $f_2(x) \geq 0$. We can therefore split mode 1 into two modes with the following invariants:

Mode 1_d: *Invariant:* $x(t) \leq T_M \quad \wedge \quad x(t) \geq (h+f)$
Mode 1_i: *Invariant:* $x(t) \leq T_M \quad \wedge \quad x(t) \leq (h+f)$

In mode 1_d the temperature x decreases; in mode 1_i it increases. Similarly, we can split mode 2 into two modes: 2_d (invariant: $T_m \leq x(t) \wedge x(t) \geq f$), in which the temperature x decreases and 2_i (invariant: $T_m \leq x(t) \wedge x(t) \leq f$), in which the temperature increases.
We want to find criticality functions $c_{1_d}, c_{1_i}, c_{2_d}, c_{2_i}$ satisfying the conditions in Definition 8. These conditions can be reformulated, using instead of the flow conditions in every mode only the monotonicity of the function x.

Positivity:
$\forall x, c_q(x) \geq 0$ *for* $q \in \{1_d, 1_i, 2_d, 2_i\}$

Flow:
$\forall x, x' \quad [(h+f \leq x \leq T_M \wedge h+f \leq x' \leq T_M \wedge x \leq x') \rightarrow c_{1_d}(x) \leq c_{1_d}(x')]$
$\forall x, x' \; [(x \leq T_M \wedge x \leq h+f \wedge x' \leq T_M \wedge x' \leq h+f \wedge x \leq x') \rightarrow c_{1_i}(x') \leq c_{1_i}(x)]$
$\forall x, x' \quad [(T_m \leq x \wedge f \leq x \wedge T_m \leq x' \wedge f \leq x' \wedge x \leq x') \rightarrow c_{2_d}(x) \leq c_{2_d}(x')]$
$\forall x, x' \quad [(T_m \leq x \leq f \wedge T_m \leq x' \leq f \wedge x \leq x') \rightarrow c_{2_i}(x') \leq c_{2_i}(x)]$

Jumps:

$c_{1_d}(T_m) \leq c_{2_d}(T_m)$	$c_{1_i}(T_m) \leq c_{2_d}(T_m)$	$c_{1_d}(T_m) \leq c_{2_i}(T_m)$	$c_{1_i}(T_m) \leq c_{2_i}(T_m)$
$c_{2_d}(T_M) \leq c_{1_d}(T_M)$	$c_{2_i}(T_M) \leq c_{1_d}(T_M)$	$c_{2_d}(T_M) \leq c_{1_i}(T_M)$	$c_{2_i}(T_M) \leq c_{1_i}(T_M)$
$c_{1_d}(h+f) \leq c_{1_i}(h+f)$	$c_{1_i}(h+f) \leq c_{1_d}(h+f)$	$c_{2_d}(f) \leq c_{2_i}(f)$	$c_{2_i}(f) \leq c_{2_d}(f)$

$\mathsf{Inv}_q(x) \rightarrow ((T_{\mathsf{min}} \leq x \wedge x \leq T_{\mathsf{max}}) \leftrightarrow c_q(x) \leq k)$, *for* $q \in \{1_d, 1_i, 2_d, 2_i\}$.

All conditions above form a set $\mathcal{K}$ of clauses which define a local theory extension. The set of clauses is satisfiable iff the set of instances in which the variables are instantiated with constants occurring in $\mathcal{K}$ is satisfiable. These constants are $T_m, f, h+f, T_M$. We use methods for constructing total models from partial ones in local theory extensions and obtain a family of functions $c_{1_d}, c_{1_i}, c_{2_d}, c_{2_i} : \mathbb{R} \rightarrow \mathbb{R}$ satisfying $\mathcal{K}$. Assume for example that the set of instances has a model in which $T_m < f \leq h+f < T_M$. Let $b_1 = h+f$ and $b_2 = f$. We can easily construct a total model for $\mathcal{K}$ by constructing for $q = 1, 2$, $c_{q_d} = c_{q_i} := c_q$, where the function c_q is strictly decreasing in the interval $(-\infty, b_q]$ and strictly increasing in the interval $[b_q, +\infty)$; $c_q(T_m) = c_q(T_M) = k$ and $c_q(b_q) = c_{q_d}(b_i) = c_{q_i}(b_q)$.

5 Conclusions

In this paper we refined methods for generating constraints on parameters which we proposed in [15,3,4] and applied then to increasingly more complex hybrid automata (parametric linear hybrid automata, parametric hybrid automata, interconnected families of hybrid automata). We then showed that we can use possibilities of automated model generation in local theory extensions for generating criticality functions in hybrid automata. We would like to continue the work analyzing possibilities of modeling systems of similar, interconnected parametric hybrid automata with a parametric number of components with a general

topology defined by using data structures, and of decomposing complex hybrid automata in order to simplify the verification tasks.

Acknowledgments. We thank Werner Damm for suggesting that simpler properties of continuous variables could be used for building criticality functions. We thank the reviewers for their helpful suggestions.

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

References

1. Alur, R., Henzinger, T.A., Ho, P.H.: Automatic Symbolic Verification of Embedded Systems. IEEE Trans. Software Eng. 22(3), 181–201 (1996)
2. Cimatti, A., Roveri, M., Tonetta, S.: Requirements Validation for Hybrid Systems. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 188–203. Springer, Heidelberg (2009)
3. Damm, W., Ihlemann, C., Sofronie-Stokkermans, V.: Decidability and complexity for the verification of reasonable linear hybrid automata. In: Proceedings of HSCC 2011, pp. 73–82. ACM (2011)
4. Damm, W., Ihlemann, C., Sofronie-Stokkermans, V.: PTIME Parametric Verification of Safety Properties for Reasonable Linear Hybrid Automata. Mathematics in Computer Science 5(4), 469–497 (2011)
5. Dolzmann, A., Sturm, T.: Redlog: Computer algebra meets computer logic. ACM SIGSAM Bulletin 31(2), 2–9 (1997)
6. Frehse, G., Jha, S.K., Krogh, B.H.: A counterexample-guided approach to parameter synthesis for linear hybrid automata. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 187–200. Springer, Heidelberg (2008)
7. Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)
8. Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On Local Reasoning in Verification. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008)
9. Ihlemann, C., Sofronie-Stokkermans, V.: On hierarchical reasoning in combinations of theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 30–45. Springer, Heidelberg (2010)
10. Platzer, A., Quesel, J.-D.: Logical verification and systematic parametric analysis in train control. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 646–649. Springer, Heidelberg (2008)
11. Platzer, A., Quesel, J.-D.: European train control system: A case study in formal verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 246–265. Springer, Heidelberg (2009)
12. Oehlerking, J., Theel, O.: Decompositional Construction of Lyapunov Functions for Hybrid Systems. In: Majumdar, R., Tabuada, P. (eds.) HSCC 2009. LNCS, vol. 5469, pp. 276–290. Springer, Heidelberg (2009)
13. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 219–234. Springer, Heidelberg (2005)

14. Sofronie-Stokkermans, V.: Efficient hierarchical reasoning about functions over numerical domains. In: Dengel, A.R., Berns, K., Breuel, T.M., Bomarius, F., Roth-Berghofer, T.R. (eds.) KI 2008. LNCS (LNAI), vol. 5243, pp. 135–143. Springer, Heidelberg (2008)
15. Sofronie-Stokkermans, V.: Hierarchical reasoning for the verification of parametric systems. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 171–187. Springer, Heidelberg (2010)
16. Taly, A., Gulwani, S., Tiwari, A.: Synthesizing switching logic using constraint solving. STTT 13(6), 519–535 (2011)
17. Wang, F.: Symbolic Parametric Safety Analysis of Linear Hybrid Systems with BDD-Like Data-Structures. IEEE Trans. Software Eng. 31(1), 38–51 (2005)

Quantifier Instantiation Techniques for Finite Model Finding in SMT*

Andrew Reynolds[1], Cesare Tinelli[1], Amit Goel[2], Sava Krstić[2], Morgan Deters[3], and Clark Barrett[3]

[1] Department of Computer Science, The University of Iowa
[2] Strategic CAD Labs, Intel Corporation
[3] New York University

Abstract. SMT-based applications increasingly rely on SMT solvers being able to deal with quantified formulas. Current work shows that for formulas with quantifiers over uninterpreted sorts counter-models can be obtained by integrating a finite model finding capability into the architecture of a modern SMT solver. We examine various strategies for on-demand quantifier instantiation in this setting. Here, completeness can be achieved by considering all ground instances over the finite domain of each quantifier. However, exhaustive instantiation quickly becomes unfeasible with larger domain sizes. We propose instantiation strategies to identify and consider only a selection of ground instances that suffices to determine the satisfiability of the input formula. We also examine heuristic quantifier instantiation techniques such as E-matching for the purpose of accelerating the search. We give experimental evidence that our approach is practical for use in industrial applications and is competitive with other approaches.

1 Introduction

Solvers for satisfiability modulo theories (SMT) are concerned with the problem of determining the satisfiability of a set of formulas in some first order theory T, which is possibly the combination of several sub-theories. SMT solvers use sophisticated and very effective techniques for deciding the satisfiability of ground formulas. While some of them can reason about quantified formulas, they do so using incomplete methods. Hence they often report "unknown" when they fail, after some predetermined amount of effort, to prove a quantified input formula unsatisfiable. For many client applications, however, it is very useful to know when such formulas are indeed satisfiable. Current SMT solvers are able to produce models of satisfiable quantified formulas only in fairly restricted cases [8], which limits their scope and usefulness. To address this limitation, in previous work we have developed a general method for efficient finite model finding in SMT [13]. More precisely, since SMT solvers work in sorted logics with both built-in and *free* ("uninterpreted") sorts, the method looks for models that interpret the latter as finite domains—and so is restricted to SMT formulas with quantifiers ranging only over the free sorts.

Like finite model finders for standard first-order logic, our method is based on checking universal quantifiers exhaustively over candidate models with increasingly large

* The work of the first two authors was partially funded by a grant from Intel Corporation.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 377–391, 2013.

domains for the free sorts, until an actual model is found. It contrasts with previous approaches for not relying on the explicit introduction of *domain constants* for the free sorts, as done by MACE-style model finders [6], and for being able to reason modulo more theories than just the theory of equality, contrary to SEM-style model finders [15]. The model finder described in [13] incorporates into a general architecture used by many SMT solvers [12] an efficient mechanism for deciding the satisfiability of a set of ground SMT formulas under finite cardinality constraints for the free sorts. This is used to find first a *candidate model*, a model $\mathcal{M}$ of a set of ground formulas generated from the input formula φ. To check that $\mathcal{M}$ satisfies φ as well, the model finder then checks, by exhaustive instantiation, that all the ground instances of φ over the universe of $\mathcal{M}$ are satisfied by $\mathcal{M}$. When this check fails, the model finder looks for a new candidate model, possibly under extended cardinality bounds for the free sorts.

Contribution. The contribution of this paper consists in two major improvements to work described in [13]: (1) a method for constructing and representing candidate models efficiently and (2) a model-based instantiation approach that avoids the explicit generation and checking of all the ground instances of the input formula. The two are strictly related since the new instantiation approach takes advantage of the way the model is represented to identify entire sets of instances that do not need to be considered.

Related Work. The data structure we use to represent candidate models is inspired by the *context* data structure introduced in the Model Evolution calculus [2]. The way we construct these models is similar to the generalization mechanism of the Inst-Gen calculus [7]. An instance generation approach similar to ours is taken by [10] for the local theory extensions method. There, the number of generated instances is reduced by finding an unsatisfiable core of relevant ground literals that are in conflict with a candidate model. A different model-based instantiation approach is followed by the Z3 SMT solver [8] where the solver itself is used as an oracle for checking the satisfiability of candidate models.

Formal Preliminaries. We work in the context of many-sorted first-order logic with equality. A (many-sorted) *signature* Σ consists of a set $\Sigma^{\mathrm{s}} \subseteq S$ of sort symbols and a set Σ^{f} of *(sorted) function symbols*, $f^{S_1 \cdots S_n S}$, where $n \geq 0$ and $S_1, \ldots, S_n, S \in \Sigma^{\mathrm{s}}$. We drop the sort superscript from function symbols when it is clear from context or unimportant. Without loss of generality, we use equality, denoted by $\approx$, as the only predicate symbol.

Given a signature Σ, well-sorted terms, atoms, literals, clauses, and formulas are defined as usual, and referred to respectively as Σ-*terms*, Σ-*atoms* and so on. A *ground term (resp. formula)* is a Σ-term (resp. formula) with no variables. A Σ-*sentence* is a Σ-formula with no free variables. Where $\mathbf{x} = (x_1, \ldots, x_n)$ is tuple of sorted variables we write $\forall \mathbf{x}\, \varphi$ as an abbreviation of $\forall x_1 \cdots \forall x_n\, \varphi$. A Σ-formula is *universal* if it has the form $\forall \mathbf{x}\, \varphi$ where φ is a quantifier-free formula.

A Σ-*structure* $\mathcal{M}$ maps each $S \in \Sigma^{\mathrm{s}}$ to a non-empty set $S^{\mathcal{M}}$, the *domain* of S in $\mathcal{M}$, and each $f^{S_1 \cdots S_n S} \in \Sigma^{\mathrm{f}}$ to a total function $f^{\mathcal{M}} : S_1^{\mathcal{M}} \times \cdots \times S_n^{\mathcal{M}} \to S^{\mathcal{M}}$. A satisfiability relation $\models$ between Σ-structures and Σ-sentences is defined as usual. A Σ-structure $\mathcal{M}$ *satisfies* (or *is a model of*) a Σ-sentence φ if $\mathcal{M} \models \varphi$. Entailment between (sets of) sentences, also denoted by $\models$, is also defined as usual.

Given a set G of ground formulas, let $\mathbf{T}_G$ be the set of all terms occurring in G. A set A of equalities and disequalities between terms in $\mathbf{T}_G$ is a *(complete) arrangement for G* if A is satisfiable and for all $s,t \in \mathbf{T}_G$ of the same sort, $s \approx t$ or $s \not\approx t$ is in A. An arrangement A for G *satisfies* G if $A \models G$. A set $E \subseteq \{s \approx t \mid s,t \in \mathbf{T}_G\}$ is a *congruence (for G)* if it is closed under entailment: for all $s,t \in \mathbf{T}_G$, $E \models s \approx t$ iff $s \approx t \in E$. The *congruence closure E^* of E (wrt. G)* is the smallest congruence for G that includes E. By construction, E^* is an equivalence relation over $\mathbf{T}_G$. For any such relation, we will assume as fixed for every sort S with terms in G, a set $\mathbf{V}^S = \{v_1^S, \ldots, v_{n_S}^S\}$ consisting of an arbitrary representatives for each equivalence class in E^*'s over terms of sort S. We call $\mathbf{V}^S$ the set of *S-values for E^** and say that terms of sort S_i *have value* v_i^S *in* E^*. For each term t we denote by $\mathsf{v}_{E^*}(t)$ its value in E^*. It can be shown (see, e.g., [1]) that E is satisfied by a structure $\mathcal{M}$ that interprets each sort S as $\mathbf{V}^S$. We call $\mathcal{M}$ a *normal model* of E. By reducing G to disjunctive normal form it is easy to show that G is satisfiable iff it is satisfied by a normal model of some set E of equalities over $\mathbf{T}_G$.

A congruence E^* over $\mathbf{T}_G$ can be uniquely extended to the arrangement $E^* \cup \{s \not\approx t \mid s \approx t \notin E^*\}$ for G. Moreover, that arrangement satisfies G whenever $E^* \models G$. So in the paper *we will often identify congruences and their associated arrangements*.

A substitution σ is a mapping from variables to terms of the same sort, such that the set $\{x \mid x\sigma \neq x\}$, the *domain* of σ, is finite. Unifying substitutions, most general unifiers (mgu's), and term variants are defined as usual. Let $\preceq$ be the usual instantiation (quasi-)ordering between terms/atoms: $s \preceq t$ iff $s\sigma = t$ for some substitution σ. If T is a set of terms and t a term, a *most-specific generalization of t in T* is any term $s \in T$ such that (1) $s \preceq t$ and (2) for all $s' \in T$ with $s \preceq s' \preceq t$, s' is a variant of s.

2 Quantifier Instantiation for Finite Model Finding

Our finite model finding method has been developed so that it can be tightly integrated into a multi-theory version of the DPLL(T) architecture [12]. A description of our model finder in terms of that architecture is provided in [13]. For the purposes of this paper, it is enough to give a high-level, stand-alone description of the basic model finding procedure restricted to general satisfiability problems (with no theories).

We can ignore the background theory T in this paper because any set of ground formulas generated by the model finder can be purified Nelson-Oppen-style into one set F_T of formulas built only with symbols of T and free constants, and one set F built only with free symbols. After adding to F_T and F a suitable set of equality constraints over their shared free constants, as prescribed by the Nelson-Oppen combination method, if F_T and F are satisfiable their respective models can be always amalgamated into a model of the original problem. Given our restriction on the quantifiers of the input problem, the model finder never needs to instantiate over the sorts of the theory T. So we can focus here just on finding models for non-theory formulas.

Without loss of generality, we consider only input problems that are the union $G \cup Q$ of a set G of ground Σ-formulas and a set Q of non-ground universal Σ-sentences for some finite signature Σ. Moreover, G contains a term of sort S for each $S \in \Sigma^{\mathrm{s}}$. We fix G, Q and Σ as above for the rest of the section.

Basic Model Finding Procedure. A basic version of the procedure, which is parametrized by a *quantifier instantiation heuristic* $\mathcal{H}$, works as follows:

1. if G is unsatisfiable return "unsat"; else, find a satisfying arrangement E^* for G
2. for each sort $S \in \Sigma^{\mathrm{s}}$, let $\mathbf{V}_S$ be the set of S-values of E^*; let $\mathbf{V} = \bigcup_{S \in \Sigma^{\mathrm{s}}} \mathbf{V}_S$
3. using $\mathcal{H}$ choose a set $I_{\mathbf{x}}$ of *valuations*, substitutions from $\mathbf{x}$ to $\mathbf{V}$, for each $\forall \mathbf{x}\, \varphi \in Q$
4. if the union of all sets $I_{\mathbf{x}}$ is empty, return "sat"; otherwise, for each $\forall \mathbf{x} \varphi \in Q$, add the instances $\{\varphi\sigma \mid \sigma \in I_{\mathbf{x}}\}$ to G, and go to Step 1

Step 1 is achieved in our model finder with a novel satisfiability solver, also described in [13], for ground formulas with finite cardinality constraints (FCC) on their sorts. If G is satisfiable, the FCC solver finds a model of G where each sort has a domain of minimal size. As in other model finders, this is done to minimize the number of possible instances of the formulas in Q. Step 2 is a by-product of the congruence closure procedure used by the FCC solver: to construct each $\mathbf{V}_S$ it is enough to collect the representatives of the congruence classes computed for S. We provide more details on this in Section 2.1. The heuristic $\mathcal{H}$ should be such that whenever $\bigcup_{\forall \mathbf{x} \varphi \in Q} I_{\mathbf{x}}$ is empty $\mathcal{M}$ satisfies Q as well. We discuss how the sets $I_{\mathbf{x}}$ are constructed in Section 2.2.

We represent normal models using the following data structure parametrized by the sets of S-values $\mathbf{V}_S$ for some arrangement A.

Definition 1 (Defining map). *Let $f^{S_1 \cdots S_n S} \in \Sigma^{\mathrm{f}}$ and let $x_1, \ldots, x_n$ be distinct variables of respective sort $S_1 \ldots, S_n$. A* defining map for *f is a finite set Δ_f of well-sorted (directed) equations of the form $f(t_1, \ldots, t_n) \approx v$ with $v \in \mathbf{V}_S$ and $t_i \in \{x_i\} \cup \mathbf{V}_{S_i}$ for $i = 1, \ldots, n$, satisfying the following requirements.*

1. *If $t_1 \approx v_1, t_2 \approx v_2 \in \Delta_f$ with $v_1 \neq v_2$ and t_1 and t_2 have an mgu σ, then σ is non-empty and $t_1\sigma = v \in \Delta_f$ for some v.*
2. *$f(x_1, \ldots, x_n) \approx v \in \Delta_f$ for some v.*

A Σ-map is a set $\Delta = \bigcup_{f \in \Sigma^{\mathrm{f}}} \Delta_f$ where each Δ_f is a defining map for f.

By construction of Δ, every *flat term*, i.e., every Σ-term $t = f(v_1, \ldots, v_n)$ with $v_1, \ldots, v_n \in \bigcup_S \mathbf{V}_S$, has exactly one most specific generalization s among the left-hand sides of equalities in Δ_f. The existence of s is guaranteed by Point 2 in Definition 1; its uniqueness by Point 1. The *value of t in Δ* is the value v in the (unique) equality $s \approx v \in \Delta_f$.

Intuitively, a Σ-map Δ represents a normal model $\mathcal{M}$ where each sort S is interpreted as the term set $\mathbf{V}_S$ and each function symbol $f^{S_1 \cdots S_n S}$ is interpreted as the function $f^{\mathcal{M}}$ mapping every $(v_1, \ldots, v_n) \in S_1^{\mathcal{M}} \times \cdots \times S_n^{\mathcal{M}}$ to the value of $f(v_1, \ldots, v_n)$ in Δ.

Proposition 1. *Let Δ be a Σ-map.*

1. *Δ induces a unique Σ-structure $\mathcal{M}_\Delta$ modulo isomorphism.*
2. *The satisfiability of universal Σ-sentences in $\mathcal{M}_\Delta$ is decidable.*
3. *Every normal model of G is induced by a Σ-map.*

We omit the simple proof of this proposition. For Point 2, we just observe that ground terms can be evaluated in $\mathcal{M}_\Delta$ bottom-up by computing the value in Δ of flat terms $f(v_1,\ldots,v_n)$. That evaluation allows one to decide ground satisfiability in $\mathcal{M}_\Delta$ in the obvious way. Since every domain of $\mathcal{M}_\Delta$ is finite, the ground satisfiability procedure can be extended to universal Σ-sentences by exhaustive instantiation of their quantifiers by all values of the corresponding sort.

We rely on normal models constructed from Σ-maps to be able to check the satisfiability of our problem $G \cup Q$ without having to generate all ground instances of its quantified formulas.

2.1 Constructing Normal Models

Given an arrangement A for the ground portion G of our input problem we wish to construct a normal model $\mathcal{M}$ of G that satisfies the quantified portion Q as well. We will refer to $\mathcal{M}$ as a *candidate model (of $G \cup Q$)*. We do this in concrete by building a Σ-map from A following a strategy that tries to maximize the number of satisfied ground instances of formulas in Q. For each function symbol f in Σ, we start building its defining map Δ_f by putting in Δ_f the equality $f(v_1,\ldots,v_n) \approx v_0$ for each term t_0 of the form $f(t_1,\ldots,t_n)$ in G where $v_i = \mathsf{v}_A(t_i)$ for $i = 0,\ldots,n$.

Collecting these equalities may produce only a partial definition for f. To complete it so that the corresponding Σ-structure satisfies G, one can use arbitrary output values for the remaining input tuples. Previous approaches such as the model-based quantifier instantiation approach implemented in the Z3 SMT solver [8] choose the same default values for all input tuples unconstrained by A. Such choices may lead to an infinite series of model checking steps and subsequent instantiations of Q if the *wrong* default values are chosen. We too use default values, but we select them in a more informed way, inspired by the instantiation heuristics used in the iProver theorem prover [11]. The main idea is to use the valuation of certain ground terms to guide the selection of default values for function symbols.

Similarly to iProver, we attempt to lift the model of a *ground abstraction* of quantified formulas. We first associate to each sort S a distinguished ground Σ-term e^S of G, which we will write ambiguously here just as e when convenient. Let σ_e be the substitution mapping all variables of sort S to e^S for each sort S. For all $f^{S_1\cdots S_n S} \in \Sigma^{\mathrm{f}}$, fix n distinct variables $x_1,\ldots,x_n$ of respective sort $S_1\ldots,S_n$. Then, for all ground Σ-terms $f(t_1,\ldots,t_n)$, let

$$f(t_1,\ldots,t_n)^\forall = f(u_1,\ldots,u_n)$$

where $u_i = x_i$ if $t_i = e$, and $u_i = \mathsf{v}_A(t_i)$ otherwise, for $i = 1,\ldots,n$. To guide the construction of a Σ-map, instead of starting with G we start with $\widehat{G} = G \cup \{\varphi\sigma_e \mid \forall \mathbf{y}\, \varphi \in Q\}$.

Once we find a satisfying arrangement A for $\widehat{G}$, we look at the values it gives to the terms containing the distinguished terms e in order to determine the choice of default values for the function symbols. As a simple example, suppose $Q = \{\forall y f(g(y)) \approx h(a,y)\}$, $\widehat{G} = G \cup \{f(g(e)) \approx h(a,e)\}$, and suppose A is a satisfying arrangement for $\widehat{G}$ such that $\mathsf{v}_A(g(e)) = v$ and $\mathsf{v}_A(h(a,e)) = u$. To complete the defining map Δ_g for g we use v as the *default* value for g, that is, we add the equation $g(x_1) \approx v$ to Δ_g. Similarly, we add the equation $h(a,x_2) \approx u$ to Δ_h. The rationale for this choice is that,

together with $f(v) \approx u$ in Δ_f, this will guarantee in this case that the corresponding normal model satisfies Q. Of course, this heuristic is not always successful in finding a satisfying model for Q right away. We describe later the corrective measures we take to find a *better* model, that is, one that falsifies fewer ground instances of formulas in Q.

The general procedure for constructing a Σ-map is the following.

Model Construction Procedure. Assuming that $\widehat{G} = G \cup \{\varphi\sigma_e \mid \forall \mathbf{y}\, \varphi \in Q\}$ is satisfiable, let A be a satisfying arrangement for it.

1. select a subset T of $\mathbf{T}_{\widehat{G}}$.
2. for each $f \in \Sigma^{\mathrm{f}}$,
   (a) let $D_1 = \{f(\mathsf{v}_A(t_1), \ldots, \mathsf{v}_A(t_n)) \approx \mathsf{v}_A(t) \mid t \in \mathbf{T}_{\widehat{G}}, t = f(t_1, \ldots, t_n)\}$
   (b) let $D_2 = \{f(t_1, \ldots, t_n)^{\forall} \approx \mathsf{v}_A(t) \mid t \in T, t = f(t_1, \ldots, t_n)\}$
   (c) let $\Delta_f = D_1 \cup D_2$ and let $\{t_i \approx v_i\}_{0 \le i \le m}$ be an arbitrary enumeration of Δ_f; for all $t_i \approx v_i$, $t_j \approx v_j$ that are unifiable with mgu σ, if $t_i\sigma$ does not already occur as a left-hand side in Δ_f, add $t_i\sigma \approx v_i$ to Δ_f
   (d) unless $f(x_1, \ldots, x_n)$ already occurs as a left-hand side in Δ_f, add $f(x_1, \ldots, x_n) \approx v$ for some arbitrary value v of the same sort
3. let $\Delta = \bigcup_{f \in \Sigma^{\mathrm{f}}} \Delta_f$ ■

Proposition 2. *The set Δ constructed by the procedure above is a Σ-map. Moreover, the Σ-structure $\mathcal{M}$ induced by Δ is a normal model of $\widehat{G}$.*

The first step of the model construction procedure is intended to choose a selection of terms containing the distinguished terms e. This selection is driven by the arrangement A itself and the way it satisfies the formulas of G. It is currently defined as follows.

Let A be a satisfying arrangement for $\widehat{G}$. For all $\psi = \forall \mathbf{y}\, \varphi \in Q$, a ground formula φ' is *selectable for* ψ if $A \models \varphi'$ and $\varphi' \models \varphi\sigma_e$. We have a strategy that chooses a selectable formula $\mathsf{sel}(\varphi)$ for each $\psi = \forall \mathbf{y}\, \varphi \in Q$ and then selects all terms in $\mathsf{sel}(\varphi)$. The set T in Step 1 of the model construction procedure is the collection of all these selected terms. The formula $\mathsf{sel}(\varphi)$ is extracted from $\varphi\sigma_e$ itself. For formulas φ in CNF it is simply a conjuction of literals, with each literal coming from a conjunct of $\varphi\sigma_e$.

Example 1. Say $Q = \{\forall y\, (f(y) \not\approx g(y) \vee h(y) \not\approx b)\}$ and

$$\widehat{G} = \{g(b) \approx a, h(a) \approx b, h(b) \approx b, a \approx f(a)\} \cup \{f(a) \not\approx g(a) \vee h(a) \not\approx b\}$$

where all terms have the same sort and $e = a$. The congruence closure E^* of the set E of equalities in $\widehat{G}$ extends to an arrangement A that satisfies $f(a) \not\approx g(a)$. With A we would select $f(a) \not\approx g(a)$ for Q's only formula, with selected terms $f(a)$ and $g(a)$.

Assuming the values of A are $\{a, b, g(a)\}$, a Σ-map constructed from A could be

$$\Delta = \{a \approx a\} \cup \{b \approx b\} \cup \{g(a) \approx g(a), g(b) \approx a, g(x_1) \approx g(a)\} \cup \{h(a) \approx b, h(b) \approx b, h(x_1) \approx b\} \cup \{f(a) \approx a, f(x_1) \approx a\}\ .$$

The Σ-structure induced by Δ *almost* satisfies Q. It does not for falsifying the instance $f(b) \not\approx g(b) \vee h(b) \not\approx b$. Adding that instance to $\widehat{G}$, we can construct an arrangement like A but with the additional value $f(b)$. This can lead to a Σ-map $\Delta' = \Delta \cup \{f(b) \approx f(b)\}$ whose induced Σ-structure does satisfy $G \cup Q$. ■

proc eval(Δ, t, σ) $\equiv$ **match** t **with**
$\quad | f(t_1,\ldots,t_n) \rightarrow$ **for** $j = 1,\ldots,n$ let $(v_j, X_j) =$ eval(Δ, t_j, σ)
choose a critical argument subset C of $\{1,\ldots,n\}$
return $(f^{\mathcal{M}_\Delta}(v_1,\ldots,v_n), \bigcup_{i\in C} X_i))$
$\quad | x \rightarrow$ **return** $(\sigma(x), \{x\})$

Fig. 1. The eval procedure. $\mathcal{M}_\Delta$ is the model induced by Δ.

2.2 Checking Models

As mentioned earlier, once we have a candidate model, i.e., a normal model $\mathcal{M}$ satisfying the ground formulas in G, a straightforward way to check that it satisfies the quantified formulas in Q is to check *all* of their ground instances over the finitely many values $\mathbf{V}$ of $\mathcal{M}$. Since a universal formula with n quantified variables each ranging over a domain of size at least k has at least n^k such instances, this is feasible in practice only when both n and k are small.

To increase the scalability of our model finding method we have developed a technique that identifies entire sets of instances satisfiable in $\mathcal{M}$ without actually generating and checking those instances individually. Since the technique is based on the model $\mathcal{M}$ (actually, on the Σ-map that represents $\mathcal{M}$), we will refer to it as the *model-based approach*, as opposed to the *naive approach* consisting of generating and checking every possible ground instances.

The main idea of the model-based approach is to determine the satisfiability in $\mathcal{M}$ of some ground instance $\varphi\sigma$ of a quantified formula $\forall \mathbf{x}\,\varphi \in Q$, generalize $\varphi\sigma$ to a whole set of F of instances equisatisfiable with $\varphi\sigma$ in $\mathcal{M}$, and then look for further instances only outside that set. The set F is computed by identifying which variables of φ actually matter in determining the satisfiability of $\varphi\sigma$. Technically, for each $\psi = \forall \mathbf{x}\,\varphi \in Q$, valuation $\sigma = \{\mathbf{x} \mapsto \mathbf{v}\}$ into $\mathbf{V}$, and ground instance $\varphi' = \varphi\sigma$ of ψ, if $\mathcal{M} \models \varphi'$ we compute a partition of $\mathbf{x}$ into $\mathbf{x}_1$ and $\mathbf{x}_2$ and a corresponding partition of $\mathbf{v}$ into $\mathbf{v}_1$ and $\mathbf{v}_2$ such that $\mathcal{M} \models \forall \mathbf{x}_2\, \varphi\{\mathbf{x}_1 \mapsto \mathbf{v}_1\}$; similarly, if $\mathcal{M} \not\models \neg\varphi'$ we compute a partition such that $\mathcal{M} \not\models \forall \mathbf{x}_2 \neg\varphi\{\mathbf{x}_1 \mapsto \mathbf{v}_1\}$. In either case, we then know that all ground instances of $\varphi\{\mathbf{x}_1 \mapsto \mathbf{v}_1\}$ over $\mathbf{V}$ are equisatisfiable with φ' in $\mathcal{M}$, and so it is enough to consider just φ' in lieu of all them. We will refer to the elements of $\mathbf{x}_1$ above as a set of *critical variables for* φ *(under* σ*)*—although strictly speaking this is a misnomer as we do not insist that $\mathbf{x}_1$ be minimal.

Checking and Generalizing Ground Instances. Treating quantifier-free formulas as Boolean terms (which evaluate to either true or false in a Σ-structure depending on whether they are satisfied by the model or not), we developed a general procedure that, given the Σ-map of a candidate model $\mathcal{M}$, a term t, and a valuation σ of t's variables, computes and returns both the value of $t\sigma$ in $\mathcal{M}$ and a set of critical variables for σ.

The procedure, defined recursively over the input term and assuming a prefix form for the logical operators as well, is sketched in Figure 1. When evaluating a non-variable term $f(t_1,\ldots,t_n)$, eval determines a *critical argument subset* C for it. This is a subset of $\{1,\ldots,n\}$ such that the term $f(s_1,\ldots,s_n)$ denotes a constant function in $\mathcal{M}$ where each s_i is the value computed by eval for t_i if $i \in C$, and is a unique variable otherwise. If f is

a logical symbol, the choice of C is dictated by the symbol's semantics. For instance, for $\approx(t_1,t_2)$, C is $\{1,2\}$; for $\vee(t_1,\ldots,t_n)$, it is $\{1,\ldots,n\}$ if the disjunction evaluates to false; otherwise, it is $\{i\}$ if t_i is the one with the best set X_i of critical variables among the elements of $\{t_1,\ldots,t_n\}$ that evaluate to true, where "best" is defined in term of another heuristic measure. If f is a function symbol of Σ, eval computes C by first constructing a custom index data structure for interpreting applications of f to values. The key feature of this data structure is that it uses information on the sets $X_1,\ldots X_n$ to choose an evaluation order for the arguments of f. For space constraints, we give just a concrete example of how this choice is made. Say eval, given the term $t = f(g(x,y,z),v_2,h(x))$, computes the values v_1,v_2,v_3 and the critical variable sets $\{x,y,z\}$, $\emptyset$, $\{x\}$ for the three arguments of f, respectively. With those sets, it will use the evaluation order $(2,3,1)$ for those arguments—meaning that the second argument is evaluated first, then the third, etc. Using the index data structure, it will first determine if $f(x_1,v_2,x_3)$ has a constant interpretation in $\mathcal{M}$. If so, then the evaluation depends on no variables and the returned set of critical variables for t will be $\emptyset$. Otherwise, if $f(x_1,v_2,v_3)$ has a constant interpretation in M, then the evaluation depends on $\{x\}$, or else it depends on the entire variable set $\{x,y,z\}$.

The next example gives more details on the whole process of generalizing a ground instance to a set of ground instances equisatisfiable with it in the given model.

Example 2. Let $Q = \{\forall y \forall z\, f(z) \approx g(y,b) \vee h(y,z) \not\approx b\}$ and $\widehat{G} = \{f(a) \approx a, f(b) \approx b, g(a,a) \approx b, h(a,a) \approx b, f(a) \approx g(a,b) \vee h(a,a) \not\approx b\}$ where a is the only distinguished ground term. Consider a Σ-map Δ constructed as in Example 1 and containing the following defining maps:

$$\Delta_g = \{g(a,b) \approx a, g(a,a) \approx b, g(x_1,b) \approx a, g(x_1,x_2) \approx b\}$$
$$\Delta_f = \{f(b) \approx b, f(a) \approx a, f(x_1) \approx a\} \qquad \Delta_h = \{h(a,a) \approx b, h(x_1,x_2) \approx b\}$$

The table below shows the bottom-up calculation performed by eval on the formula $\varphi = f(z) \approx g(y,b) \vee h(y,z) \not\approx b$ with Δ above and $\sigma = \{y \mapsto a, z \mapsto a\}$.

input	output	critical arg. subset	input	output	critical arg. subset
z	$(a,\{z\})$	//	$h(y,z)$	$(b,\emptyset)$	$\emptyset$
y	$(a,\{y\})$	//	$f(z) \approx g(y,b)$	$(\mathsf{true},\{z\})$	$\{1,2\}$
b	$(b,\emptyset)$	//	$h(y,z) \not\approx b$	$(\mathsf{false},\emptyset)$	$\emptyset$
$f(z)$	$(a,\{z\})$	$\{1\}$	$f(z) \approx g(y,b) \vee h(y,z) \not\approx b$	$(\mathsf{true},\{z\})$	$\{1\}$
$g(y,b)$	$(a,\emptyset)$	$\{2\}$			

For most entries in the table the evaluation is straightforward. For a more interesting case, consider the evaluation of $g(y,b)$. First, the arguments of g are evaluated, respectively to $(a,\{y\})$ and $(b,\emptyset)$, but with evaluation order $(2,1)$. After evaluating y to b, using an indexing data structure built from Δ_g for the evaluation order $(2,1)$, eval is able to quickly determine that the term $g(x_1,b)$ has constant value a for all x_1. Hence it returns an empty set of critical variables for $g(y,b)$.

Similarly, the fact that eval returns $(\mathsf{true},\{z\})$ for the original input formula φ and the valuation $\sigma = \{y \mapsto a, z \mapsto a\}$, means that it was able to determine that all ground instances of $\varphi\{z \mapsto a\} = (f(a) \approx g(y,b) \vee h(y,a) \not\approx b)$, not just the instance $\varphi\sigma$, are satisfied in $\mathcal{M}$. Our model finder can then use this information to completely avoid generating and checking those instances. ■

proc choose_instances$(\Delta, \varphi, \mathbf{x})$ $\equiv$
  $I_{\mathbf{x}} := \emptyset$; $t_{next} := \mathbf{v}_{min}$ where $\mathbf{v}_{min}$ is the minimum of $\mathbf{V}_{\mathbf{x}}$
  do
    $t := t_{next}$
    $(v, \{x_{i_1}, \ldots, x_{i_m}\}) := \mathsf{eval}(\Delta, \varphi, \{\mathbf{x} \mapsto t\})$
    if $v = \mathsf{false}$ **then** $I_{\mathbf{x}} := I_{\mathbf{x}} \cup \{\{\mathbf{x} \mapsto t\}\}$
    $t_{next} := \mathsf{next}_i(t)$ where i is the minimum of $\{i_1, \ldots, i_m, n+1\}$
  while $t_{next} \neq t$
  return $I_{\mathbf{x}}$

Fig. 2. The choose_instances procedure. We assume that $\mathbf{x} = (x_1, \ldots, x_n)$.

Collecting Ground Instances. For any given quantified formula ψ, the eval procedure allows us to identify a set of instances over $\mathbf{V}$ that can be represented by a single one, as far as satisfiability in the candidate model $\mathcal{M}$ is concerned. The next question then is how to generate a set I of instances that together represent *all* instances of ψ over $\mathbf{V}$ that are falsified by $\mathcal{M}$. This kind of exhaustiveness is crucial because it allows us to conclude correctly that $\mathcal{M} \models \psi$ by just checking that I is empty.

We present a procedure that relies on eval for computing the set I above or, rather, a set of valuations for generating the elements of I from ψ. The procedure is fairly unsophisticated and quite conservative in its choice of representative instances, which makes it very simple to implement and prove correct. Its main shortcoming is that it does not take full advantage of the information provided by eval, and so may end up producing more representative instances than needed in many cases. The development of a more selective procedure is left to future work.

Let $\psi = \forall \mathbf{x}\, \varphi \in Q$ with $\mathbf{x} = (x_1, \ldots, x_n)$. For $i = 1, \ldots, n$, let S_i be the sort of x_i and let $\mathbf{V}_{\mathbf{x}} = \mathbf{V}_{S_1} \times \cdots \times \mathbf{V}_{S_n}$. For each $S \in \{S_1, \ldots, S_n\}$, let $<_S$ be an arbitrary total ordering over the values $\mathbf{V}_S$ of sort S. Let $<$ be the *reversed lexicographic*[1] extension of these orderings to the tuples in $\mathbf{V}_{\mathbf{x}}$ and observe that $\mathbf{V}_{\mathbf{x}}$ is totally ordered by $<$.

For every $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbf{V}_{\mathbf{x}}$ let $\mathbf{v}[i]$ denote the i^{th} element of $\mathbf{v}$ and let $\mathsf{next}_i(\mathbf{v})$ denote the smallest tuple $\mathbf{u}$ wrt. $<$ such that $\mathbf{v}[j] <_{S_j} \mathbf{u}[j]$ for some $j \geq i$, if such tuple exists, and denote $\mathbf{v}$ itself otherwise (including when $i > n$). For instance, with $n = 3$, $S_1 = S_2 = S_3$ and $\mathbf{V}_{S_1} = \{a, b\}$ with $a <_{S_1} b$, we have that $\mathsf{next}_2(a,a,a) = (a,b,a)$, $\mathsf{next}_2(a,b,a) = (a,a,b)$, $\mathsf{next}_3(a,a,b) = (a,a,b)$, and $\mathsf{next}_3(a,b,b) = (a,b,b)$. Note that $\mathbf{v} \leq \mathsf{next}_i(\mathbf{v})$ for all $\mathbf{v}$.

The instantiation heuristic $\mathcal{H}$ used in the model finding procedure presented in Section 2 is implemented by the procedure choose_instances described in Figure 2, which takes in a quantifier-free formula φ with variables $\mathbf{x}$ and returns a set $I_{\mathbf{x}}$ of valuations σ for $\mathbf{x}$ such that $\mathcal{M} \not\models \varphi\sigma$. At each execution of its loop the procedure implicitly determines with eval a set of I of instances of φ that are equisatisfiable with $\varphi\{\mathbf{x} \mapsto \mathbf{v}\}$ in $\mathcal{M}$, where $\mathbf{v}$ is the tuple stored in the program variable t. The next value t_{next} for t is a greater tuple chosen to maintain the invariant that all the tuples between t and t_{next} generate instances of φ that are in I. To see that, it suffices to observe that these tuples differ from t only in positions that correspond to non-critical variables of φ, namely those before

[1] This is defined similarly to the standard lexicographic extension except that the last component of a tuple is the most significant one, then the last but one, and so on.

position i where x_i is the first critical variable of φ in the enumeration $x_1, \ldots, x_n$. This observation is the main argument in the proof of the following result.

Proposition 3. *Let* $\mathbf{v}_0, \ldots, \mathbf{v}_m$ *be all values successively taken by the variable* t *in the loop of* choose_instances. *Let* v_{max} *be the maximum element of* $\mathbf{V}_\mathbf{x}$. *Then for all* $i = 1, \ldots, m$,

1. $\mathbf{v}_{i-1} < \mathbf{v}_i$,
2. *for all* $\mathbf{u}$ *with* $\mathbf{v}_{i-1} \leq \mathbf{u} < \mathbf{v}_i$, $\mathcal{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{u}\}$ *iff* $\mathcal{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{v}_{i-1}\}$,
3. *for all* $\mathbf{u}$ *with* $\mathbf{v}_m \leq \mathbf{u} \leq \mathbf{v}_{max}$, $\mathcal{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{u}\}$ *iff* $\mathcal{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{v}_m\}$.

For this proposition it follows immediately that $\mathcal{M} \models \forall \mathbf{x} \varphi$ if and only if the set $I_\mathbf{x}$ returned by choose_instances$(\Delta, \varphi, \mathbf{x})$ is empty.

We remark that, for our model finding purposes, there is no need for the procedure choose_instances to compute the full set $I_\mathbf{x}$ once it contains at least one valuation. Any non-empty subset would suffice to trigger a (more incremental) revision of the current candidate model $\mathcal{M}$. That said, our current implementation does compute the whole set and adds all the corresponding instances to Q before recomputing another model for it. Our initial experiments show that computing and using one valuation at a time is worse for overall performance than computing and using the full set $I_\mathbf{x}$.

2.3 Enhancements Based on Heuristic Instantiation

Many SMT solvers rely on heuristic instantiation methods for finding unsatisfiable instances for quantified formulas. These methods typically use *E-matching* techniques [3] to generate heuristically relevant instances, which are based on matching distinguished terms, called *triggers*, with ground terms in the problems. We found that E-matching can be helpful in our model finder as well, even for satisfiable problems.

Enhanced Model Finding Procedure. Our original heuristic $\mathcal{H}$ from Section 2 for quantifier instantiation can be enhanced with E-matching to a heuristic $\mathcal{H}'$ as follows:

1. choose a set of triggers T_ψ for each $\psi \in Q$, and return valuations based on E-matching for (T_ψ, G)
2. if no such instances exist, apply the original $\mathcal{H}$.

Applying E-matching helps the model finder detect the unsatisfiability of its input formulas more promptly in cases where a conflict is easily identifiable. Furthermore, it may also accelerate the discovery of a model for satisfiable input problems, since the instances it generates can help rule out bad choices of candidate models more quickly.

Recall that in the basic model finding procedure, quantifier instantiation is applied *after* finding a model of the ground formulas G of minimal size. By waiting to apply quantifier instantiation until after model minimization, we also avoid pitfalls common to E-matching-based procedures such as, for instance, *matching loops* where certain terms get generated at every instantiation round. Since only a finite number of terms exist for a given cardinality bound on a sort, our approach guarantees that E-matching will eventually rule out the given bound, or terminate with no instances produced.

Most E-matching techniques generate triggers automatically, but in fairly uninformed ways, typically choosing every applicable term (or set of terms) in a quantified formula ψ as a trigger. In our model finding method, the selection heuristic described in Section 2.1 can be used as a criterion for trigger generation by using first as triggers the terms that were selected for the construction of the current candidate mode. The intuition is that if we are basing the satisfiability of ψ on the default values given for a function symbol f, then we need only be concerned with possible exceptions to those defaults. Our current implementation follows this criterion.

3 Experimental Results

The model finding method introduced in [13] is implemented within the CVC4 SMT solver. For the present work, we implemented the naive and the model-based approach for quantifier instantiation as alternative configurations of CVC4's finite model finder. We ran experimental comparisons for these approaches on three sets of benchmarks.

First we considered formulas derived from verification conditions generated by DVF [9], a tool used at Intel for verifying properties of security protocols and design architectures, among other applications, comparing configurations of the model finder against CVC4 in native mode (i.e., not using the model finder) and Z3 version 4.1, which we previously found to be the best SMT solver besides CVC4 on these benchmarks [13]. Second, we considered benchmarks from the latest version of the TPTP library (5.4.0), comparing against various automated theorem provers and model finders for first order logic, as well as the two SMT solvers above. Third, we considered a set of SMT benchmarks translated from proof obligations generated by the Isabelle prover, comparing again with CVC4 in native mode and Z3.[2]

In all experiments we used revision 4751 of CVC4 1.0, both in native mode (indicated here as **cvc4**) and in finite model finding mode. The default configuration of the latter (**cvc4+f**) applies naive quantifier instantiation as described in Section 2.2, and no heuristic instantiation. The other model finding configurations use either model-based quantifier instantiation as described in Section 2.2 (**cvc4+fm**), or just heuristic quantifier instantiation as described in Section 2.3 (**cvc4+fi**), or both (**cvc4+fmi**).

The first set of experiments was run on a Linux machine with an 8-core 2.60GHz Intel® Xeon® E5-2670 processor. The second and third on a cluster of 5 identical Linux machines with a 2.4 GHz AMD Opteron 250s and 2 GB of available memory.

Intel Benchmarks. We considered 3 of the 5 classes of benchmarks from [13]; the other two are uninteresting as their problems can be solved quickly by **cvc4+f**. The **agree** class is from [14] while the **apg** and **bmk** classes are verification conditions internal to Intel. The benchmarks contain a variety of SMT theories, including arithmetic, arrays, datatypes, free functions over free sorts and built-in sorts, but with quantifiers limited to free sorts. Both unsatisfiable and satisfiable benchmarks were considered, the latter produced by manually removing necessary assumptions from verification conditions. The results are summarized in Figure 3 for various configurations of CVC4 and for

[2] The finite model finder, detailed results, and the non-proprietary benchmarks discussed in this section are available at `http://cvc4.cs.nyu.edu/experiments/CADE24-2013/` .

	Sat						Unsat					
	agree (15)		**apg** (17)		**bmk** (31)		**agree** (139)		**apg** (124)		**bmk** (83)	
Solver	solved	time	solved	time	solved	time	solved	time	solved	time	solved	time
z3	0	0	0	0	0	0	**139**	**3.5**	**124**	**9.0**	83	2.5
cvc4	0	0	0	0	0	0	135	2.9	124	10.0	**83**	**1.7**
cvc4+f	15	13.7	16	199.4	30	1200.1	127	3772.4	118	2243.3	81	1496.5
cvc4+fi	**15**	**12.3**	17	492.2	30	829.5	139	185.8	122	338.8	83	656.7
cvc4+fm	15	21.3	**17**	**209.9**	31	374.2	122	5007.5	120	1114.9	81	827.3
cvc4+fmi	15	13.6	17	220.6	**31**	**175.5**	139	183.4	122	336.6	83	664.9

Fig. 3. Results for DVF benchmarks. All times are in seconds. Best performances are in bold font.

Z3. We show results for the 412 problems from the previous study that were non-trivial for CVC4's model finder.[3] All configurations had a 600s timeout per problem.

For the satisfiable benchmarks, CVC4's model finder is the only tool able to solve at least one. Additionally, through use of model-based quantifier instantiation, it is now able to solve all of them within the timeout. Moreover, the best configuration of the model finder, **cvc4+fmi**, solves each benchmark within 60s.

For the unsatisfiable benchmarks, Z3 is the overall winner, solving all of them within the timeout. Pairing heuristic quantifier instantiation with finite model finding (configurations with **cvc4+*i***) is beneficial, as it even solves four problems that **cvc4** cannot solve. We found that each unsatisfiable problem can be solved by either **cvc4** or **cvc4+fmi**, and in less than 3s. Configuration **cvc4+fmi** solves all unsatisfiable benchmarks within 900s, suggesting that CVC4's model finder makes consistent progress towards answering unsatisfiable on provable DVF verification conditions. Also, **cvc4+fmi** is an order of magnitude faster than **cvc4+f** on unsatisfiable benchmarks solved by each of them. From the perspective of verification tools, the results here seem promising. A feasible strategy for discharging a verification condition would be to first use an SMT solver hoping that it can quickly find it unsatisfiable with E-matching techniques; and then resort to finite model finding if needed to either answer unsatisfiable, or produce a model representing a concrete counterexample for the verification condition.

TPTP Benchmarks. For these benchmarks we also compared against Paradox [6] and iProver [11]. Paradox is a MACE-style model finder that uses preprocessing optimizations such as sort inference and clause splitting, among others, and then encodes to SAT the original problem together with increasingly looser constraints on the size of the model. iProver is an automated theorem prover based in the Inst-Gen calculus that can also run in finite model finding mode (**iprover-fm**). In that mode, it incrementally bounds model sizes in a manner similar to MACE-style model finding. However, it encodes the whole problem into the EPR fragment, for which it is a decision procedure. Since these two tools are limited to classical first-order logic with equality, we considered only the unsorted first-order benchmarks of TPTP.

The results for a 30s timeout per benchmark, are shown in Figure 4. CVC4's model finder with exhaustive instantiation (**cvc4+f**) can find 975 benchmarks to be satisfiable. That number goes up to 1025 with model-based quantifier instantiation (**cvc4+fm**). While better than Z3, which finds 888 satisfiable benchmarks, our model finder still trails the overall performance of the other provers on these problems. Paradox, the best

[3] The rest are solved in less than 0.5s by all configurations of the model finder.

	paradox	iprover	iprover-fm	z3	cvc4	cvc4+f	cvc4+fm	cvc4+fmi
Sat	**1344**	995	1231	888	33	975	1025	955
Unsat	1272	5556	383	**5934**	5295	2633	2754	3028

Fig. 4. Results for 15561 benchmarks taken from the TPTP library, with a 30s timeout. Of these benchmarks, 1995 are known to be satistiable, and 12586 are known to be unsatisfiable.

here, finds 1344 satisfiable benchmarks. We attribute this to the fact that we have implemented none of the advanced preprocessing techniques, such as sort inference and clause splitting, that have been shown to be critical for finding finite models of TPTP benchmarks. Nevertheless, CVC4's model finder is capable solving a handful of benchmarks that neither Paradox nor iProver can solve. In particular, it solves two satisfiable benchmarks with 1.0 difficulty rating, which means that no known ATP system had solved these problems when version of 5.4.0 of the TPTP library was released.

Figure 4 shows also results for unsatisfiable problems. Although these results are not comparable to those achieved by state-of-art theorem provers, such as Vampire and E, we note that Z3 solves the most benchmarks, 5924. Interestingly, an additional 35 unsatisfiable problems with difficulty rating 1.0 were found in this study by Z3, **cvc4**, iProver and **cvc4+fmi**, which respectively solve 21, 6, 4, and 1 of these uniquely.

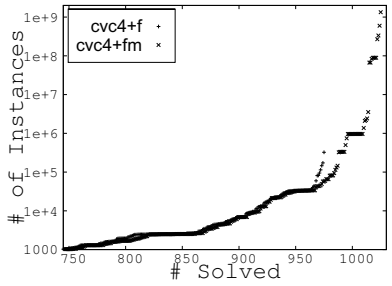


Fig. 5. Comparison of satisfiable problems found with and without model-based quantifier instantiation. A point (x, y) on this graph says the configuration solves x benchmarks each with a total of at most y ground problems of quantified formulas.

To further evaluate the impact of model-based quantifier instantiation on our model finder, we recorded statistics on the domain size of quantified formulas in benchmarks solved by its various configurations. We measured the total number of instances for all quantified formulas occurring in a problem (a quantified formula over n variables each with domain size k has n^k instances). For a problem with d total instances, the configuration **cvc4+f** must explicitly generate these d instances, while a model-based configuration may avoid doing so. For these experiments, **cvc4+f** was only able to solve 4 problems having more than 100k instances, the maximum having around 325k instances. On the other hand, **cvc4+fm** was capable of solving 41 problems having more than 100k instances, with the largest having more than 1.3 billion instances. This information is plotted in Figure 5, showing how the model-based instantiation approach improves the scalability of our model finder and allows it to solve benchmarks where exhaustive instantiation is clearly infeasible. We stress that model finders such as Paradox have other ways of handling the explosion in the number of instances, namely by minimizing the number of variables per clause. We expect that coupling these

Sat	Arrow_Order	FFT	FTA	Hoare	NS_Shared	QEpres	StrongNorm	TwoSquares	TypeSafe	TOTAL
z3	3	19	24	46	10	49	1	17	11	180
cvc4	0	9	0	0	0	0	0	8	0	17
cvc4+f	22	138	172	153	56	79	**12**	59	69	760
cvc4+fm	**26**	139	171	151	49	80	**12**	59	69	756
cvc4+fmi	**26**	**151**	**174**	**159**	**60**	**81**	**12**	**60**	**78**	**801**
Unsat	**Arrow_Order**	**FFT**	**FTA**	**Hoare**	**NS_Shared**	**QEpres**	**StrongNorm**	**TwoSquares**	**TypeSafe**	**TOTAL**
z3	**261**	**224**	**765**	**497**	**135**	236	**240**	451	**325**	**3134**
cvc4	199	217	682	456	97	**244**	231	**486**	239	2851
cvc4+f	120	99	298	214	36	105	84	316	132	1404
cvc4+fm	102	91	330	246	26	117	80	310	128	1430
cvc4+fmi	155	170	467	328	42	161	97	411	188	2019

Fig. 6. Results for Isabelle Benchmarks. Numbers of problems solved within 30s.

techniques with the model-based techniques used here will lead to additional improvements in the scalability of our model finder.

Isabelle Benchmarks. Recent work has shown that SMT solvers, in particular Z3, are effective at discharging Isabelle proof obligations whose encoding can be represented with theories [5]. Model finding can be useful in Isabelle for debugging and for brute-force proof minimization [4]. More generally, it is useful to interactive theorem provers that are based on heuristically selecting a set of relevant background axioms which might be sufficient to prove a conjecture. In this case, a model finder could be used to quickly identify axiom sets that are not large enough for a given conjecture.

We considered a set of 13,041 benchmarks generated from Isabelle and kindly provided by Sascha Böhme. The benchmarks in this set correspond to both provable and unprovable conjectures.[4] Most of them contain quantifiers, and a significant portion contain integer arithmetic. For many, quantifiers are limited to the free sorts, thus making our finite model finding approach applicable. Since CVC4 does not yet have support for non-linear arithmetic, we report results only for the 11,187 benchmarks that do not contain non-linear arithmetic constraints. Additionally, CVC4 ignored various hints (such as weight values) that were given to Z3 for quantifier instantiation.

The results are shown in Figure 6. In these experiments, using E-matching accelerates the search for models, as **cvc4+fmi** finds more satisfiable problems (810) than both **cvc4+f** (760) and **cvc4+fm** (756). All configurations of CVC4's model finder find many more satisfiable problems than Z3, which finds only 180 of them overall. For unsatisfiable problems, Z3 is the overall winner, solving 3,134 of them, followed by the **cvc4** configuration with 2,851. Interestingly, while **cvc4+fmi** solves only 2,019 unsatisfiable benchmarks, 244 of them are not solved by Z3, and 164 are not solved by **cvc4**.

4 Conclusion

We have introduced a few quantifier instantiation techniques for finite model finding in SMT which drastically improve the scalability of our basic model finding procedure and are useful in various applications. Our experiments show that our model-based quantifier instantiation approach is useful for finding models where exhaustive instantiation is infeasible, and can be improved further by integrating heuristic instantiation in it, especially for unsatisfiable problems.

[4] It is our understanding that these benchmarks are a superset of those discussed in [5].

Future research includes improvements to the instance generation technique in Section 2.2, and further generalizing the approach to the construction of models for built-in theories. We are currently investigating ways to modify the selection heuristics of Section 2.1 to generate candidate models (in some fragments) of the theory of arrays. We plan to investigate further approaches for finding models of formulas with quantifiers ranging over built-in domains such as the integers.

Acknowledgements. We would like thank Sascha Böhme for providing the Isabelle benchmarks and François Bobot for his help in writing a TPTP front end for CVC4.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
2. Baumgartner, P., Tinelli, C.: The Model Evolution calculus as a first-order DPLL method. Artificial Intelligence 172, 591–632 (2008)
3. de Moura, L., Bjørner, N.S.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007)
4. Blanchette, J.C.: Personal communication (2013)
5. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 116–130. Springer, Heidelberg (2011)
6. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model building. In: CADE-19 Workshop: Model Computation – Principles, Algorithms, Applications, pp. 11–27 (2003)
7. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. In: Proceedings of LICS 2003, pp. 55–64. IEEE Computer Society (2003)
8. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
9. Goel, A., Krstic, S., Tuttle, R.L.M.: SMT-based system verification with DVF. In: Proceedings of SMT 2012 (2012)
10. Jacobs, S.: Incremental instance generation in local reasoning. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 368–382. Springer, Heidelberg (2009)
11. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (System description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
12. Krstić, S., Goel, A.: Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720, pp. 1–27. Springer, Heidelberg (2007)
13. Reynolds, A., Tinelli, C., Goel, A., Krstić, S.: Finite model finding in SMT. In: Proceedings of CAV 2013. LNCS. Springer (accepted, 2013)
14. Tuttle, M.R., Goel, A.: Protocol proof checking simplified with SMT. In: Proceedings of NCA 2012, pp. 195–202. IEEE Computer Society (2012)
15. Zhang, J., Zhang, H.: SEM: a system for enumerating models. In: Proceedings of IJCAI 1995, pp. 298–303 (1995)

Automating Inductive Proofs Using Theory Exploration

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone

Department of Computer Science and Engineering,
Chalmers University of Technology
{koen,moa.johansson,danr,nicsma}@chalmers.se

Abstract. HipSpec is a system for automatically deriving and proving properties about functional programs. It uses a novel approach, combining theory exploration, counterexample testing and inductive theorem proving. HipSpec automatically generates a set of equational theorems about the available recursive functions of a program. These equational properties make up an algebraic specification for the program and can in addition be used as a background theory for proving additional user-stated properties. Experimental results are encouraging: HipSpec compares favourably to other inductive theorem provers and theory exploration systems.

1 Introduction

We are studying the problem of automatically proving algebraic properties of programs. Our aim is to build a tool that programmers can use to support software development. This paper describes current progress towards this goal, in particular addressing the problem of automating inductive proofs.

We work in a subset of the strongly typed functional programming language Haskell. Our subset consists of monomorphic, terminating programs without type classes or primitive types (like `Int`). The only data types are algebraic data types, functions and uninterpreted types. Removing these restrictions is ongoing work.

There are two key advantages of using Haskell as the input language. Firstly, a pure functional programming language is semantically simpler and thus easier to reason about than languages with side effects. Secondly, many Haskell programmers already use QuickCheck [5], a tool for property-based random testing, which means that many Haskell program are already annotated with formal properties (tested, but not proved).

The main obstacles one encounters when doing automated verification of functional programs are (1) when and how to apply induction, and (2) how to discover auxiliary lemmas or generalisations which may be required in inductive proofs. Let us look at a simple example. Consider the following Haskell program implementing the list reverse function in two different ways, `rev` and `qrev`. The latter uses a helper function `revacc` with an accumulating parameter which leads to a function with better time complexity. Their definitions are:

```
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 392–406, 2013.

```
revacc []     acc = acc
revacc (x:xs) acc = revacc xs (x:acc)

qrev xs = revacc xs []
```

A natural property one would like to verify is that the functions above produce the same result: ∀ `xs`. `rev xs = qrev xs`. Suppose we attempt to prove this by structural induction on `xs`. This will fail as the inductive hypothesis `rev as = qrev as` is too weak to prove `rev (a:as) = qrev (a:as)`. What is needed here is an additional lemma such as `rev xs++ys = revacc xs ys`, from which the original conjecture follows as a special case when `ys` happens to be the empty list. This is a typical example of the kind of generalisations which are required in proofs about functions with accumulator variables. One of the main challenges for inductive theorem provers is how to discover such lemmas automatically.

Current inductive theorem provers such as IsaPlanner [8], Zeno [19] and ACL2 [13] support a simple lemma discovery technique called *lemma calculation*, by which a new lemma is suggested by replacing some common subterm in a stuck goal by a variable. Although this technique works very well for many proofs, it is not enough for the above example, which cannot be automatically proved by these systems. The now defunct CLAM proof-planner had in addition a so-called *proof-critic* for discovering more complex generalisations [9], such as the one required in the example, but only if other basic lemmas were given by the user.

Our approach differs from the *top-down* manner in which the above systems work. Instead of waiting for the proof to somehow get stuck, we use *bottom-up* lemma discovery, or *theory exploration*. Our tool, called HipSpec, gets its name from its two subsystems which we developed previously: the automated inductive prover Hip [18], and the conjecture generation system QuickSpec [6]. Hip tries to prove a conjecture by enumerating all possible ways of doing structural induction over the free variables, and then calling an automated first-order prover to prove them. QuickSpec creates thousands of terms involving the functions of a given API, and computes equivalence classes over these terms by means of testing. Each pair of terms t_1, t_2 in an equivalence class gives rise to a conjecture $t_1 = t_2$.

HipSpec reads in a program, but besides trying to tackle any of the user-given properties, it asks QuickSpec to produce a list of conjectures about the program. HipSpec then sends these conjectures to Hip and those that are proved can be used as lemmas in subsequent proof-attempts. After this theory exploration phase, the properties stated by the programmer are tried, using all the proved lemmas as background theory.

There are several theory exploration systems which have been applied to discover theorems in inductive theories [12,15,16], but none have been fully integrated with an automated theorem prover in order to supply the prover with lemmas. Instead, these systems simply generate and prove a set of 'interesting' equations summarising the main properties about the program, which are then presented to the user. In fact, HipSpec may also be used in this manner without any user-stated properties.

Let us return to the example property about `rev`. HipSpec calls QuickSpec, which within a few seconds conjectures a set of equations about the functions involved. HipSpec feeds these to Hip, which tries to prove them. Those that can be proved without induction are redundant and can be discarded; the lemmas needing induction are shown below[1]:

No	Conjecture		Proved using[2]
(*1*)	`xs++[]`	`= xs`	`xs`
(*2*)	`(xs++ys)++zs`	`= xs++(ys++zs)`	`xs`
(*3*)	`rev xs++rev ys`	`= rev (ys++xs)`	`ys`, (*1*), (*2*)
(*4*)	`revacc (revacc xs ys) []`	`= revacc ys xs`	`xs`
(*5*)	`revacc (revacc xs ys) zs`	`= revacc ys (xs++zs)`	`xs`
(*6*)	`revacc xs ys++zs`	`= revacc xs (ys++zs)`	`zs`, (*1*), (*2*), (*5*)
(*7*)	`revacc xs (rev ys)`	`= rev (ys++xs)`	`xs`, (*1*), (*3*), (*6*)

The original property is now easily proved: it follows directly from (*7*), letting `ys = []`, and the definition of `qrev`; induction is not even needed. Note that lemma (*4*) is not needed for proving the original property. Discovering some unnecessary lemmas is a (potentially disadvantageous) side-effect of the bottom-up approach.

Contributions. We augment the automated induction landscape with a new method which uses a bottom-up theory exploration approach to find auxiliary lemmas. This approach combines our own earlier work on conjecture generation based on testing (QuickSpec) and induction principle enumeration (Hip). By adding proof capabilities on top of QuickSpec we also get a system which can be used as a stand-alone theory exploration system.

Our hypothesis is that:

1. Algebraic equations constructed from terms up to a certain depth form a rich enough background theory for proving many algebraic properties about programs without specialised proof-critics.
2. A reasoning system for functional programs can be built on top of an automatic first-order theorem prover.
3. A system combining (1) and (2) can be used both as a theorem prover and as an efficient theory exploration system, producing background lemmas comparable to those appearing in human-created libraries.

The experimental results in this paper have so far confirmed this.

2 Implementation

Below we describe in more detail how Hip and QuickSpec work, and how they are combined in HipSpec.

[1] The variables are implicitly universally quantified over total and finite values.
[2] This column shows the induction variables and which lemmas were used.

2.1 Hip

Hip [18] is an automatic tool for proving user-stated equality or implicational properties about Haskell programs. Hip starts by compiling the definitions in the program at hand to first-order logic. For each property stated in the program, it systematically applies different induction rules, yielding first-order proof obligations, which are tested for validity using off-the shelf automated first-order theorem provers. If one proof obligation succeeds, the original conjecture was valid. Thus, the first-order prover takes care of non-inductive reasoning, while Hip adds inductive reasoning at the meta-level. In the context of HipSpec, Hip is configured to apply structural induction up to a given depth on one or more variables. Hip, however, also supports co-inductive proof techniques such as fixed point induction. The focus of our work in HipSpec is currently not on proving termination, so we restrict ourselves by allowing only well-founded definitions, and put the responsibility on the end user to enforce this policy for now.

2.2 QuickSpec

QuickSpec [6] conjectures equations about a functional program by means of testing. The user of QuickSpec provides a list of functions and their types, a random test data generator for each of the types involved, a set of variables (usually 2-3 per type), and a term depth limit (usually 3). QuickSpec starts by creating a set of terms, called the *universe*, consisting of all well-typed terms built from the functions and variables given, whose depth is within the given limit. It then partitions this universe into equivalence classes by running a finite number of random tests (usually 100); two terms will be in the same equivalence class if and only if they were equal for all tests. This equivalence relation in turn gives rise to a huge set of conjectured equations about the tested program (typically thousands or tens of thousands). For the sake of human users, QuickSpec also includes a final phase which prunes away equations that follow from simpler ones, leaving only a small core of equations from which all original equations follow. This core is usually presented to the user (usually 10-25 equations). However, when HipSpec uses QuickSpec to generate lemmas, it does not use the pruning phase, because valuable lemmas may be pruned away. For example, even when an equation E_1 implies a more complex equation E_2, we can not necessarily discard E_2, because E_2 may be provable by induction whereas E_1 may not be. In fact, E_2 may very well be needed as a lemma to prove E_1! So, HipSpec considers the full set of equations produced by QuickSpec before pruning.

2.3 HipSpec

HipSpec's operation is illustrated in Figure 1. We start by running QuickSpec on the program source file, which generates a list of conjectures. We also translate the program source code to a first-order theory using Hip.

HipSpec maintains three sets of equations: *active conjectures*, which we still need to consider, *failed conjectures*, which we have already tried to prove but

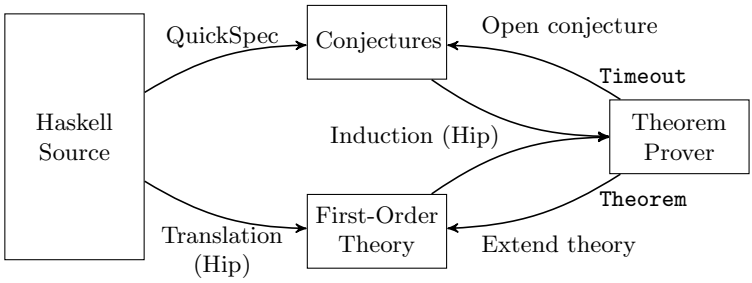


Fig. 1. An overview of HipSpec

failed, and *lemmas*, which we have managed to prove. The *first-order theory* in Figure 1 consists of Hip's translation of our program plus the current set of lemmas. Initially the active conjectures consist of all equations that QuickSpec found (even those that would have been removed by pruning), and the failed conjecture set and lemma set are empty.

The main loop works as follows:

1. Pick a conjecture c from the active conjecture set (using a heuristic described below).
2. Check if c follows from the lemmas found so far by equational reasoning only. If so, discard c, and re-iterate.
3. Otherwise, ask Hip to prove the conjecture by induction, using definitions and previously proved lemmas as background theory.
4. If Hip succeeds, move c to the lemma set, and move some failed conjectures back to the active conjectures (based on a heuristic described below).
5. If Hip does not succeed within a set timeout, move c to the failed conjectures.

The loop ends when the active conjecture set is empty.

Picking the conjecture. The performance of HipSpec completely depends on one heuristic: which active conjecture to try to prove next. Our current heuristics are rather crude; more sophisticated techniques are further work.

Our basic strategy is to prove simpler equations before more complicated ones. We define simplicity as follows. A smaller term is simpler than a bigger term; if two terms have the same size, the term with more variables is simpler (because it might be more general). For example, `(x+y)+z=x+(y+z)` is simpler than `(x+x)+y=x+(x+y)`. The simplicity of an equation $t_1 = t_2$ is determined by whichever of t_1 and t_2 is the most complex.

We also take into account the call graph of the program. For example, if we are proving properties about the natural numbers, we prove as much as possible about `+` before starting on `*`, since `*` calls `+`. More precisely, when choosing which conjecture to prove next, we pick the one whose call graph is the smallest; if two conjectures have the same size call graph, we pick the simplest one.

Discarding trivial consequences. It is quite expensive to send every conjecture to Hip to be proved, when we may have thousands of them. Luckily, QuickSpec has a lightweight theorem prover based on congruence closure. This prover can efficiently answer questions of the form "given these lemmas, can I prove this equation?", replying either "yes" or "don't know".

Whenever we pick a conjecture, we check if this prover can prove it from the current lemmas without induction. If so, we just discard it. This filters out most trivial conjectures that are provable without induction.

Re-activating failed conjectures. When we prove a lemma, we sometimes move some failed conjectures back to the active conjectures. HipSpec's rule is to wait until the set of active conjectures is empty and then move *all* failed conjectures back to the active set, provided that at least one new lemma was proved since last attempting the conjecture. This guarantees termination.

We have experimented with more elaborate heuristics in this step, eagerly adding failed conjectures back. These heuristics can help in certain examples, but so far none have been sufficiently general. Perhaps surprisingly, the simple method described above works well for all examples in this article. More sophisticated heuristics are further work.

3 Examples

This section gives examples of successful proofs and their related theory explorations, as well as an example showing some current limitations of our approach.

3.1 Rotating the Length a of List

This simple property of the `rotate` function is surprisingly difficult to prove[3]:

```
prop_rotate xs = rotate (length xs) xs =:= xs
```

The `rotate` function takes a natural number `n` and returns the list resulting from removing the `n` first elements and appending them to the end. Rotating a list by its length returns the original list. Although this property is very simple to state it is surprisingly hard to prove by mathematical induction, as it requires a generalised version to be proved, which implies `prop_rotate`. This generalisation itself can be proved by induction.

Given the standard definitions of append, length and Peano numbers with successor `S` and zero `Z`, and the below definition of `rotate`, HipSpec finds and proves such a generalisation, and uses it to prove `prop_rotate`:

```
rotate Z     xs     = xs
rotate (S n) []     = []
rotate (S n) (x:xs) = rotate n (xs ++ [x])
```

[3] Here, `=:=` is HipSpec's notation for equality.

The lemmas for which HipSpec needed induction are in Figure 2. Lemma (*8*) is the required generalisation, from which it proves `prop_rotate`, which follows as a special case when `ys` is the empty list. Notice that lemma (*8*) itself requires lemmas (*1*) and (*2*). A number of additional lemmas are also discovered, which are not of use in this particular proof, but could well be useful in other proofs. The whole process of theory exploration and the proof of `prop_rotate` took 17 seconds, with less than a second spent in QuickSpec and the rest of the time spent in various proofs of the generated equations.

No	Conjecture			Proved by
(*1*)	`xs++[]`	`=`	`xs`	`xs`
(*2*)	`(xs++ys)++zs`	`=`	`xs++(ys++zs)`	`xs`
(*3*)	`rotate n (rotate m xs)`	`=`	`rotate m (rotate n xs)`	`n`, `m`
(*4*)	`rotate (S n) (rotate m xs)`	`=`	`rotate (S m) (rotate n xs)`	`xs`, (*3*)
(*5*)	`rotate n [x]`	`=`	`[x]`	`n`
(*6*)	`length (xs++ys)`	`=`	`length (ys++xs)`	`xs`, `ys`
(*7*)	`length (rotate n xs)`	`=`	`length xs`	`n`, (*6*)
(*8*)	`rotate (length xs) (xs++ys)`	`=`	`ys++xs`	`xs`, (*1*), (*2*)
(*9*)	`rotate (length xs) xs`	`=`	`xs`	(*8*)

Fig. 2. Properties generated and proved about the theory of lists with `++`, `rotate`, and `length`. The third column shows which induction variables and lemmas were used.

As this proof requires both generalisation and lemma discovery it was identified in 2005 as an automated reasoning challenge beyond the capabilities of state-of-the-art reasoning systems ([3], p. 77). We are not aware of any other theorem provers which prove this theorem fully automatically, without the help of user-supplied lemmas.

3.2 Nicomachus' Theorem

Using Peano arithmetic, with standard definitions of addition and multiplication recursively on the first argument, we will try to get HipSpec to prove Nicomachus' Theorem. This states that the sum of the n first cubes is the nth triangle number squared: $\sum_{k=1}^{n} k^3 = (\sum_{k=1}^{n} k)^2$. We define two functions: `tri` calculates triangle numbers and `cubes n` calculates the sum of the first `n` cubes.

```
tri Z     = Z                cubes Z     = Z
tri (S n) = tri n + S n      cubes (S n) = cubes n + (S n*S n*S n)
```

Using these definitions, Nicomachus' theorem is stated as follows:

```
prop_Nicomachus x = cubes x =:= tri x * tri x
```

When HipSpec is given the definitions of plus, multiplication, `tri` and `cubes`, it generates and proves (by induction) the properties listed in Figure 3 below, which takes 10 seconds. The properties are listed in the order they were proved. In (*8*)

No	Conjecture	Lemmas used	Induction on
(*1*)	`x+y = y+x`		`x, y`
(*2*)	`x+(y+z) = (y+x)+z`	(*1*)	`z`
(*3*)	`x*y = y*x`	(*2*)	`x, y`
(*4*)	`x*(y*z) = (y*x)*z`	(*1*), (*2*), (*3*)	`x, y`
(*5*)	`x*(y+y) = y*(x+x)`	(*1*), (*2*), (*3*), (*4*)	`y`
(*6*)	`(x*y)+(x*z) = x*(y+z)`	(*1*), (*2*), (*3*)	`z`
(*7*)	`tri x*(y+y) = (x*y)*S x`	(*1*), (*2*), (*3*), (*4*), (*6*)	`x`
(*8*)	`tri x+tri x = x+(x*x)`	(*1*), (*2*), (*3*)	`x`
(*9*)	`tri x*tri x = cubes x`	(*1*), (*2*), (*3*), (*6*), (*8*)	`x`

Fig. 3. Properties proved about the theory with natural number addition, multiplication, triangle numbers (`tri`) and sum of cubes (`cubes`)

the well-known identity $\sum_{k=1}^{n} k = n(n+1)/2$ is proved, using previously proved lemmas. From this lemma HipSpec proves Nicomachus' Theorem in (*9*). Due to the order in which HipSpec ends up proving the conjectures in this example, some unnecessary lemmas are included in figure 3, e.g. (*5*) and (*7*).

3.3 Insertion Sort Produces a Sorted List

Currently, QuickSpec can only generate equational lemmas. To prove that, for example, insertion sort produces a sorted list requires conditional lemmas. We state this property as `prop_sorted xs = sorted (isort xs) =:= True`.

In order to prove `prop_sorted` we need the conditional lemma `sorted xs ==> sorted (insert x xs)`, where `insert` is the sorted list insertion function used by `isort`, but HipSpec only can only discover and prove the somewhat peculiar equations (lemmas *1-4*) in Figure 4. HipSpec also discovers, but fails to prove, some additional properties (conjectures *5-9*). For example, property (*5*), which states that `insert` is commutative in its first argument. These equations are not proved because they require conditional lemmas.

Although not proved, QuickSpec has tested these equations and not found a counterexample. Hence, even a failed proof attempt may at least give some insight into the properties of the program. The runtime for this example was 8 seconds.

No	Conjecture		Induction on
(*1*)	`x<=x`	`= True`	`x`
(*2*)	`x<=S x`	`= True`	`x`
(*3*)	`S x<=x`	`= False`	`x`
(*4*)	`insert y (x:[])`	`= insert x (y:[])`	`x, y`
(*5*)	`insert x (insert y xs)`	`= insert y (insert x xs)`	
(*6*)	`sorted (insert x xs)`	`= sorted xs`	
(*7*)	`isort (insert x xs)`	`= isort (x:xs)`	
(*8*)	`sorted (isort xs)`	`= True`	
(*9*)	`isort (isort xs)`	`= isort xs`	

Fig. 4. Results for the theory of insertion sort. Properties *1-4* were proved, while properties *5-9* were not, as they require conditional lemmas.

4 Evaluation

HipSpec has two modes of use. Firstly, it can be used as an automated induction to prove user-given conjectures using theory exploration to find necessary lemmas. In this case, the individual lemmas that are discovered in the background and used in the proofs are of less interest for the user, since the focus is on proving the user supplied properties automatically. Theory exploration is treated more like a black box.

Secondly, HipSpec can be used in a more speculative manner, as a standalone theory exploration system. In this case, the user expects HipSpec to discover and prove a set of basic equational properties about the given program. Here it becomes important not to swamp the user with trivial or overly complicated equations. Rather, we wish to present the user with a concise set of elegant equations summarising the main properties, much like the libraries in proof assistants such as Isabelle. The hope is that these may be useful in later interactive reasoning or as an algebraic specification of the program.

The examples come from the theorem proving literature and assume terminating functions over total values. We used Z3 [7] as a backend for HipSpec in these experiments. As the program is translated to a first order theory, we did not use any of Z3's built-in theories or decision procedures, but we did exploit its support for types and constructor functions. The source code for HipSpec and all experimental results are available online [1,2].

4.1 HipSpec as a Theorem Prover

HipSpec was evaluated on two test suites from the inductive theorem proving literature. The test suites consist of conjectures about natural numbers, lists and binary trees. As they feature a large number of unrelated functions, HipSpec was run separately for each property. This reduces the number of generated equations because HipSpec will ignore any function that is not (directly or indirectly) reachable from the property. It also means that HipSpec cannot use already-proved properties from the test suite to prove later ones. Thus, the order of the properties in the test suite does not matter: they are proved independently.

HipSpec was configured to give a timeout of 1 second for each individual proof obligation sent to the prover, and to allow induction on up to two variables simultaneously using one-step structural induction.

Test Suite A consists of 85 conjectures with both first- and higher-order functions about lists, natural numbers and binary trees [10]. These were originally formalised for the IsaPlanner system in Isabelle's HOL and have since been translated into other formalisms to compare the Zeno and ACL2 Sedan provers [19,4] and the Dafny system [14]. As these systems use different logics we note that the functions are not defined in exactly the same way in the different experiments. This test suite was originally designed for evaluating IsaPlanner's rippling heuristic in the presence of if- and case-expressions, which are expressed as higher-order functions in Isabelle, and cause trouble for IsaPlanner's syntax-based rippling heuristic. Hence, from a lemma discovery point

of view, many proofs are rather easy: 67 theorems can be proved without extra lemmas, and 12 do not require induction. The results for the different provers on the 85 conjectures are summarised below:

HipSpec	Zeno [19]	ACL2s [4]	IsaPlanner [10]	Dafny [14]
80	82	74	47	45

HipSpec performs well, with the majority of failures being due to proofs requiring conditional lemmas, as HipSpec only is able to generate equations. For one property (number 81), we had to configure HipSpec to use induction on three variables; this is counted as a success in the table above. Zeno performs best, failing only on three examples, two fewer than HipSpec. However, HipSpec can prove two theorems that Zeno cannot: `rev (drop i xs) = take (len xs-i) (rev xs)` and `rev (take i xs) = drop (len xs-i) (rev xs)`.

Test Suite B consists of 50 theorems about lists and natural numbers and was previously used to demonstrate proof-critics in the CLAM prover [9], which is unfortunately no longer maintained. As opposed to Test Suite A, most theorems here do require auxiliary lemmas, generalisations, case-splits or non-standard inductions. CLAM proves 41 of the 50 theorems fully automatically. The remaining 9 theorems were proved interactively. They require generalisation (including the `rev` example from §1 and the `rotate` example from §3.1), for which CLAM needed the help of some user-supplied lemmas. Again, HipSpec was not given any auxiliary lemmas. Fully automatically, it proved 44 theorems, including 6 of the 9 theorems which CLAM proved with the help of user-supplied lemmas.

We managed to prove 3 further theorems (properties 33–35) by adjusting HipSpec's settings. These three properties concern accumulating versions of multiplication, factorial and exponentiation. Because we are using Peano arithmetic, these functions return large results, and the testing phase used too much memory: we supplied a flag that causes QuickSpec to compare results up to some size bound, so results that are too large will be considered equal. There were also too many conjectures, so we added a flag to limit the *size* of the generated terms. We did *not* have to give any lemmas by hand. In total, HipSpec proved 47 theorems, including the 9 for which CLAM needed user-supplied lemmas.

We also tested Zeno on these examples: it can prove 21, but not any of the ones requiring complex generalisations.

Finally, we remark that the bottom-up approach taken by HipSpec is naturally a bit slower than IsaPlanner and Zeno, which typically perform proofs in less than a second. Most successful proof attempts are very fast, with the long runtimes arising from cases with a lot of failed proof attempts.

For test suite A, all properties required less than a minute on a normal desktop computer [1]. The vast majority required less than 15 seconds, and most 1–2 seconds. For test suite B, the 44 successful properties required at most 15 seconds, most of them 1–2 seconds. The three properties for which we needed to tweak the settings ranged from 30 seconds to 40 minutes. Of the three failed properties, two took about five minutes before giving up, the third 8 seconds.

As mentioned, HipSpec may also discover some superfluous lemmas not strictly required for the proof of the user-stated property. In these examples, there are

very few such lemmas and the theorem prover's performance was not notably affected by these being added to the theory.

4.2 HipSpec as a Theory Exploration System

In these experiments HipSpec is given a program as an input, without any user-properties stated. The aim is to present the user with a concise set of equational properties that have been discovered and proved. We exploit the pruning algorithm already implemented in QuickSpec to achieve this. QuickSpec was originally built as a standalone system for suggesting algebraic specifications of programs using testing. When used on its own, it prunes the many equations it generates by heuristically ordering them and removing those that trivially follow from previous ones. We refer to [6] for a detailed description of this pruning algorithm. When HipSpec is used in theory exploration mode, it first attempts to prove as many conjectures as we can, just as in the theorem-proving mode. Then it takes the set of the conjectures that it proved, or that trivially follow from what it proved, and applies the pruning algorithm to this set. As a result, HipSpec often produces a smaller and more concise set of lemmas than it does when used in theorem-prover mode. The final list of equations does not depend on what order we proved things in, or on what needed induction, only on what the theory implies.

We have applied HipSpec to some simple theories from the theory exploration literature [12,16], one about natural numbers, with `+` and `*`, and three small theories about lists: 1) `append`, `reverse` and `length`, 2) `append`, `reverse` and `map` and 3) `append`, `foldl` and `foldr`. The theorems produced are presented in Figure 5. HipSpec generates these theorems much faster than IsaCoSy and IsaScheme: it takes only between 6-12 seconds for each theory (full results available online [1]), while IsaCoSy and IsaScheme may require hours. We expect this to be due to the congruence closure reasoning of QuickSpec, which reduces the search space and integrates counterexample checking in the term generation phase.

We also perform the same precision-recall analysis as in [12,16] to assess the quality of the generated theories using Isabelle's libraries[4] as reference. This experiment assumes that the Isabelle library is so well-designed that it contains exactly all interesting properties and nothing more. The results are summarised in Table 1, where *recall* measures how many of the theorems in the library were also produced by HipSpec, and *precision* measures how many of the theorems HipSpec produced were also in the library, i.e. how well it avoids producing "superfluous" theorems.

HipSpec performs very well: for the lists, it generates all theorems in Isabelle's library, plus theorem *L3* in Figure 5, which is the closest we can get to the useful lemma `length (xs ++ ys) = length xs + length ys` since we did not include the `+` operator in the program. For the natural numbers, HipSpec fails to generate three of the library theorems: the standard formulations of associativity are missing (instead HipSpec generates two variants in theorems *N5* and *N6*) and

[4] `http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/`

Natural Numbers

N1.	`x+y = y+x`	*N6.**	`x*(y*z) = y*(x*z)`
N2.	`x*y = y*x`	*N7.*	`x+S y = S (x+y)`
N3.	`x+Z = x`	*N8.*	`x*S y = x+(x*y)`
N4.	`x*Z = Z`	*N9.**	`x*(y+y) = y*(x+x)`
*N5.**	`x+(y+z) = y+(x+z)`	*N10.*	`(x*y)+(x*z) = x*(y+z)`

Lists

L1.	`xs++[] = xs`
L2.	`(xs++ys)++zs = xs++(ys++zs)`
*L3.**	`length (xs++ys) = length (ys++xs)`
L4.	`length (rev xs) = length xs`
L5.	`rev (rev xs) = xs`
L6.	`rev xs++rev ys = rev (ys++xs)`
L7.	`map f xs++map f ys = map f (xs++ys)`
L8.	`map f (rev xs) = rev (map f xs)`
L9.	`foldl f (foldl f x xs) ys = foldl f x (xs++ys)`
L10.	`foldr f (foldr f x xs) ys = foldr f x (ys++xs)`

Fig. 5. Theory exploration results: theorems generated by HipSpec. Theorems marked by $*$ were not in Isabelle's library.

the theorem `S x + y = x + S y` is excluded. However, all three can be trivially derived by equational reasoning from the theorems HipSpec does produce.

Table 1. Theory Exploration results. Note that IsaScheme was evaluated on a natural number theory also including exponentiation [16].

	HipSpec	IsaCoSy [12]	IsaScheme [16]	Isabelle
#Thms Naturals	10	16	16*	12
Precision	80%	63%	100%*	-
Recall	73%	83%	46%*	-
#Thms Lists	10	24	13	9
Precision	90%	38%	70%	-
Recall	100%	100%	100%	-

5 Related Work

Inductive theories do not allow cut-elimination and are thus undecidable. In practice, this means that auxiliary lemmas (themselves requiring an inductive proof) may be required to complete a proof. Inductive theorem provers which support some form of automated lemma discovery, such as ACL2's induction tactic [4], CLAM [9], IsaPlanner [8] and Zeno [19], use a *top-down* approach by which lemmas are discovered from failed proof-attempts. HipSpec differ from all of these in its *bottom-up* theory exploration approach. HipSpec automatically tries to discover a background theory for the relevant functions, building up something like the human-created lemma libraries available for interactive provers such

as Isabelle [17] or ACL2 [13]. Experimental evaluation shows that HipSpec's bottom-up approach compares well in terms of finding the right lemmas. Some types of lemmas are difficult to discover in the top-down approach, for instance the generalised version needed to prove the theorem `rev xs = qrev xs []`, and many other similar theorems featuring accumulator variables. While the CLAM system could discover the `rev`/`qrev` generalisation given some other basic lemmas, HipSpec discovers it all automatically. Zeno, IsaPlanner and ACL2 do not support this type of lemma discovery at all, and thus fail on theorems of this kind. In HipSpec, there is always a risk of discovering extra irrelevant lemmas too. However, these may perhaps be useful in other proofs.

Both CLAM and IsaPlanner are based on the rippling heuristic for guiding rewriting of the step-case towards the inductive hypothesis. The advantage of rippling is that it guarantees termination of rewriting, and that rewrite rules may be used both ways around if need be. Rippling is a syntax-based heuristic, which may cause problems for instance on conjectures where a lot of case-analysis is required, as highlighted by Test Suite A in §4 where HipSpec, Zeno and ACL2 performed better than the rippling-based IsaPlanner. HipSpec relies on an off-the-shelf prover as backend which has no termination guarantee like rippling-based provers. Instead termination is enforced by using a timeout, which means that there is a risk of missing proofs which just take a little bit too long. When special-purpose rippling-based provers fail, the user may inspect the final proof state to see where the proof got stuck. HipSpec cannot currently.

While most other provers have some form of built-in rewriting tactics, HipSpec and the program verifier Dafny [14] instead send proof obligations to external automated provers. Like HipSpec, Dafny applies induction on the meta-level and passes the resulting proof obligations to the theorem prover Z3, which was also used as a backend for HipSpec in the experiments in this article. Dafny does not, however, support automated lemma discovery, so auxiliary lemmas must be supplied by the user. The obvious advantage is that off-the-shelf automated provers are often very fast and powerful. However, as the provers are treated as black boxes we do not get a readable proof, or any information if a proof fails. IsaPlanner checks proof steps in Isabelle and can produce readable output of complete or partial proofs. Zeno can output proofs in Isabelle format, which can then be re-checked in the proof assistant, ensuring correctness. Readable and checkable proofs are further work in HipSpec.

HipSpec is the only system which can be used both as an inductive theorem prover and as a theory exploration system. The IsaCoSy and IsaScheme theory explorers were developed for automating the creation of lemma libraries for inductive theories in Isabelle [12,16]. Both systems use IsaPlanner to prove conjectures that pass counterexample checking, but differ in the heuristics they use to generate conjectures. Experiments in which the outputs of IsaCoSy were *manually* fed back to IsaPlanner have been successfully performed [11]. However, neither is fully integrated with the theorem prover: IsaPlanner cannot call either of these *automatically* while proving user-given properties, In contrast, HipSpec is fully automatic. Both IsaCoSy and IsaScheme are considerably slower than HipSpec, although all three systems produce similar sets of lemmas.

6 Conclusion and Further Work

HipSpec is an automated inductive theorem prover and a theory exploration system. It takes a novel bottom-up approach to lemma discovery by using theory exploration to first build a richer background theory in which user-given properties are proved. In experimental evaluation, HipSpec performs very well in comparison with other systems: in particular, it succeeds in proving theorems about tail-recursive functions that require generalisations, which no other system can prove fully automatically without user-supplied lemmas. HipSpec also performs very well as a standalone theory exploration system, producing sets of lemmas with high precision and recall when compared to Isabelle's libraries. Furthermore, it does so in seconds rather than hours like previous systems.

Ultimately, we would like to use HipSpec in a tool for automatically proving properties of Haskell programs, making it usable by "normal" programmers, much like the popular QuickCheck tool [5]. In order to extend HipSpec to the full Haskell language we need to add support also for infinite and lazy data-structures and non-terminating functions in QuickSpec and in HipSpec's property language. The Haskell-to-FOL translation system HALO [20] already supports this, and Hip supports co-inductive reasoning and fixpoint induction. The theory-exploration machinery does however need to be extended to record which lemmas hold for all values of a type (including partial ones) and which ones only hold for completely-defined total values.

Another area of further work is providing user feedback from failed proofs, and producing checkable proofs. It could interesting to experiment with a different prover backend, from which information about failed proof attempts can be reclaimed, rather than treating the prover as a black box.

References

1. HipSpec evaluation results, `http://www.cse.chalmers.se/~danr/hipspec`
2. HipSpec source code repository, `http://www.github.com/danr/hipspec`
3. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: meta-level guidance for mathematical reasoning. Cambridge University Press (2005)
4. Chamarthi, H.R., Dillinger, P., Manolios, P., Vroon, D.: The ACL2 Sedan theorem proving system. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 291–295. Springer, Heidelberg (2011)
5. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of ICFP, pp. 268–279 (2000)
6. Claessen, K., Smallbone, N., Hughes, J.: QuickSpec: Guessing formal specifications using testing. In: Proceedings of TAP, pp. 6–21 (2010)
7. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Dixon, L., Fleuriot, J.D.: Higher order rippling in IsaPlanner. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 83–98. Springer, Heidelberg (2004)

9. Ireland, A., Bundy, A.: Productive use of failure in inductive proof. Journal of Automated Reasoning 16, 79–111 (1996)
10. Johansson, M., Dixon, L., Bundy, A.: Case-analysis for rippling and inductive proof. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 291–306. Springer, Heidelberg (2010)
11. Johansson, M., Dixon, L., Bundy, A.: Dynamic Rippling, Middle-Out Reasoning and Lemma Discovery. In: Siegler, S., Wasser, N. (eds.) Walther Festschrift. LNCS, vol. 6463, pp. 102–116. Springer, Heidelberg (2010)
12. Johansson, M., Dixon, L., Bundy, A.: Conjecture synthesis for inductive theories. Journal of Automated Reasoning 47(3), 251–289 (2011)
13. Kaufmann, M., Panagiotis, M., Moore, S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
14. Leino, K.R.M.: Automating induction with an SMT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 315–331. Springer, Heidelberg (2012)
15. McCasland, R., Bundy, A., Autexier, S.: Automated discovery of inductive theorems. In: Matuszewski, R., Rudnicki, P. (eds.) From Insight to Proof: Festschrift in Honor of A. Trybulec (2007)
16. Montano-Rivas, O., McCasland, R., Dixon, L., Bundy, A.: Scheme-based theorem discovery and concept invention. Expert Systems with Applications 39(2), 1637–1646 (2012)
17. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
18. Rosén, D.: Proving Equational Haskell Properties using Automated Theorem Provers, MSc. Thesis, University of Gothenburg (2012)
19. Sonnex, W., Drossopoulou, S., Eisenbach, S.: Zeno: An automated prover for properties of recursive data structures. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 407–421. Springer, Heidelberg (2012)
20. Vytiniotis, D., Rosén, D., Peyton Jones, S., Claessen, K.: HALO: Haskell to logic through denotational semantics. In: Proceedings of POPL 2013. ACM (2013)

E-MaLeS 1.1

Daniel Kühlwein[1], Stephan Schulz[2], and Josef Urban[1]

[1] Radboud University Nijmegen, Nijmegen, Netherlands*
[2] Technische Universität München, München, Germany

Abstract. Picking the right search strategy is important for the success of automatic theorem provers. E-MaLeS is a meta-system that uses machine learning and strategy scheduling to optimize the performance of the first-order theorem prover E. E-MaLeS applies a kernel-based learning method to predict the run-time of a strategy on a given problem and dynamically constructs a schedule of multiple promising strategies that are tried in sequence on the problem. This approach has significantly improved the performance of E 1.6, resulting in the second place of E-MaLeS 1.1 in the FOF divisions of CASC-J6 and CASC@Turing.

1 Introduction

Automatic theorem provers (ATPs) for first-order logic search for proofs of a conjecture in a potentially infinite space of derivations. Experience has shown that no single search strategy can be expected to perform well over very diverse proof problems. Thus, most theorem provers provide dozens or even hundreds of parameters. For systems based on modern equational calculi, parameters include clause selection schemes, term orderings, inference and reduction rules used, etc.

The theorem prover E [8] uses a language for describing useful combinations of parameters as *strategies*. Such strategies can be automatically evaluated over large problem sets and compiled directly into C source code. Over 200 strategies have been named and evaluated for E 1.6, and the best of them are included in the E source code.

This large number of strategies with associated performance data suggests the use of data-driven methods to estimate how to solve new problems. E is one of the the first ATPs that have applied machine learning to strategy selection. Below, we first describe how the *automatic mode* of E is generated using simple analogy-based learning. We then introduce an alternative method based on kernel-based learning that automatically determines a schedule of E strategies. The resulting strategy-scheduling meta-system, E-MaLeS 1.1, outperforms the E's original single-strategy automated mode.

2 Learning of Strategy Selection

Characterizing ATP problems is a hard problem. An ambitious formulation could be for example:

* The authors were supported by the NWO projects "MathWiki a Web-based Collaborative Authoring Environment for Formal Proofs" and "Learning2Reason".

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 407–413, 2013.

Given a set of ATP strategies and a large class of ATP problems problems, find a (typically finite) set of efficiently computable problem features *that can be used to efficiently partition the set of problems into subclasses that share a common best strategy.*

The set of features is in practice suggested by the intuition of the ATP implementer or other domain expert (for example, a mathematician who might know what features could be important for distinguishing different classes of problems in his domain). Some of the features can correspond to precise knowledge about logical calculi, for example, if the problem is Horn, or effectively propositional. Others can express hunches, for example, if there are many or few clauses, symbols, terms, etc. If the set of problems uses symbols consistently (this is the case for recent large-theory corpora created from the Mizar and Isabelle libraries, and for the SUMO and Cyc common-sense ontologies), then the (combinations of) symbols (and derived structures, like terms) are often also relevant for proof search.

This paper presents two learning methods that have been developed to estimate the best strategy for a new problem. Both are based on the results of strategies run on the TPTP problems [12]. The traditional one by Stephan Schulz is an instance of the *case-based reasoning* method. The newer alternative uses *kernel-based* learning. Both methods rely on the TPTP being a sufficiently representative set of ATP problems of different kinds, however the methods can be applied for any sufficiently big corpus of problems (e.g. the MPTP [14]). Both methods require a relevant *feature characterization* of problems and a record of the performance of different strategies for training. Once trained, both provide suggestions for strategies to use on new problems, either a single strategy for the original method, or a schedule of strategies for the newer method.

2.1 E's Feature Characterization

The set of features used in E for problem characterization is listed in Table 1. All features apply to the clausal form of a problem. A clause is called *negative* if it only has negative literals. It is called *positive* if it only has positive literals. A ground clause is a clause that contains no variables. In this setting, we refer to all negative clauses as "goals", and to all other clauses as "axioms". Clauses can be *unit* (having only a single literal), *Horn* (having at most one positive literal), or *general* (no constraints on the form). All unit clauses are Horn, and all Horn clauses are general. Goals have no positive literals and are hence always at least Horn.

2.2 E's Automatic Mode

E supports an automatic mode that analyzes the problem and determines all major search parameters (literal selection function, clause selection heuristics, term ordering, and a number of mostly discrete options controlling optional simplifications and preprocessing steps). It has been conservatively extended from the very first implementation in E 0.3 *Castleton*, released in 1999.

Table 1. Problem features used by strategy selection in E

Feature	Description
`axioms`	Most specific class (unit, Horn, general) describing all axioms
`goals`	Most specific class (unit, Horn) describing all goals
`equality`	Problem has no equational literals, some equational literals, or only equational literals
`non_ground_units`	Number of unit axioms that are not ground
`ground_goals`	Are all goals ground?
`clauses`	Number of clauses
`literals`	Number of literals
`term_cells`	Number of terms (including subterms)
`ground_positive_axioms`	Number of positive axioms that are ground
`max_fun_arity`	Maximal arity of a function or predicate symbol
`avg_fun_arity`	Average arity of symbols in the problem
`sum_fun_arity`	Sum of arities of symbols in the problem
`clause_max_depth`	Maximal clause depth

The automatic mode is based on a static partitioning of the set of all CNF problems into disjoint classes. It is generated in two steps. First, the set of all training examples (typically the set of all current TPTP problems) is classified into disjoint classes using the features listed in Table 1. For the numeric features, threshold values have originally been selected to split the TPTP into 3 or 4 approximately equal subsets on each feature. Over time, these have been manually adapted using trial and error.

Once the classification is fixed, a Python program reads the different classes and a set of test protocols describing the performance of different strategies on all problems from the test set. It assigns to each class one of the strategies that solves the most examples in this class. For *large* classes (arbitrarily defined as having more than 200 problems), it picks the strategy that also is fastest on that class. For small classes, it picks the globally best strategy among those that solve the maximum number of problems. A class with zero solutions by all strategies is assigned the overall best strategy.

2.3 Strategy Scheduling

Strategy learning is currently being used by E's automatic mode to predict the best strategy for a problem. The predicted strategy is then run for the full time.

An alternative approach (known mainly from Gandalf [13], E-SETHEO [11], and Vampire [7]) is strategy scheduling. The idea is to run different strategies for fractions of the overall time. This can help when the strategies are sufficiently orthogonal (solve different problems), and the problem classification is not good enough to clearly point to one best strategy.

In the simplest setting, the suitable set of strategies is considered independent of the problem features. In that case, given a database of results of many strategies on a large set of problems (TPTP, MPTP, etc.), the goal is to cover the set of solvable problems by as few strategies as possible. This is an NP-complete problem, which however seems to be quite easy (for our instances) for systems like MiniSat++ [10]. For example, for an experiment with covering randomly chosen 400 problems from the MPTP2078 benchmark by 280 E strategies, MiniSat++ can find the minimal cover (9 strategies) in 0.09 s.

We present a learning based method that depending on the problem features predicts the time each strategy needs to solve the problem. The predictions are used to schedule in which order and for how long the strategies should be run.

2.4 E-MaLeS 1.1

E-MaLeS (E Machine Learning of Strategies) uses the kernel-based MOR algorithm, see [1] for details. E-MaLeS is freely available at `http://cs.ru.nl/~kuehlwein/`. E-MaLeS learns a function that predicts the performance for each strategy on each problem. Given the features of a problem defined in Table 1, E-MaLeS predicts for each strategy s how long E running s will need to solve the problem. A similar approach has successfully been used in the SAT community [16].

The MOR algorithm is an instance of kernel-based learning. Kernel-based learning is a machine learning approach that finds (typically non-linear in the features) approximations of the training data by minimizing a *loss function* describing the difference between learned approximation and training data. By mapping the data in a higher-dimensional vector space, kernel methods combine the expressiveness of a high-dimensional function space with the simplicity of linear regression. Intuitively, kernels can be seen as a similarity measure between different data points (in our case problems). Kernel-based methods are among the most successful algorithms applied to various problems from bioinformatics to information retrieval to computer vision [9].

E-MaLeS uses a Gaussian kernel. The similarity measure induced by this kernel can be imagined as a Gaussian distribution around each data point with the width σ of the distribution being an adjustable parameter. To ensure that the learned functions generalize well, a regularization parameter λ is used. Regularization adds an additional term based on the complexity of the learned function to the loss function. The more complex a function, the bigger the penalty. The intuition behind this is that complex function are more likely to overfit. The value of λ determines the weight of the regularization term, with $\lambda = 0$ being equivalent to no regularization.

The values for σ and λ are determined via a 10-fold cross-validation. First, we define logarithmically scaled grids of potential values. The training dataset is then shuffled and divided into two parts using a 70/30 split. For each parameter pair the algorithm trains a function based on the data points in the larger part and evaluates it (i.e. compares the predicted run times with the actual run times) on the data points in the smaller part. This process is repeated 10 times. The

parameter pair with the best average performance (i.e. the minimum average squared difference between the predicted run times and the actual run times) on the smaller set is then used for the final learning. The goal of cross-validation is to estimate the performance of the learned function on unseen data points.

During the learning phase, the input to E-MaLeS is a list of strategies and a list of problems with their features, together with the performances of the strategies on the problems within a fixed time limit (e.g. 300 seconds). The features are first normalized to values between 0 and 1. Ideally, we would have the exact time needed until a proof is found for each problem-strategy pair. Unfortunately, real world limitations restrict us to finite run times – in our case 300 seconds. Problems that were not solved within this time limit are ignored. Note that this leads to different training data for different strategies and a bias towards lower times. Thus, the learned prediction functions are likely to predict a time that is lower than the actual needed time. An alternative to simply ignoring unsolved problems would be to use a large fixed time for each (e.g. 600 seconds). However, in initial experiments this did not show any improvement.

When trying to solve a new problem, E-MaLeS employs a combination of E's automatic mode and strategy scheduling. First, the automatic mode is run for 60 seconds.[1] If the automatic mode fails to find a proof, the features of the problem are computed and normalized.[2] For each strategy the time needed to solve the problem is predicted using the prediction function learned during the setup. Since E's timeout parameter expects seconds, the predicted times are rounded up to next full second. The strategies are then run for their predicted time, starting with the strategy with the smallest predicted time.[3] If the sum of the rounded predicted times is less than the total time given, the remaining free time is spread equally over all strategies.

3 Results

Both E-MaLeS 1.1 and E 1.6 competed at CASC@Turing and CASC-J6. In both competition, E-MaLeS 1.1 won the second place in the FOF division, solving more problems than E 1.6. The results are shown in Tables 2 and 3

In the FOF division of CASC@Turing, E-MaLeS solved 4.6% more problems than E. If we only compare the results of the new problems, E-MaLeS solved 14.5% more problems. The results for FOF division of CASC-J6 are similar, with E-MaLeS solving 4% more problems than E. On the new problems, E-MaLeS solved 13.2% more problems than E.

[1] Running the auto mode first allows us to reduce the number of training examples of the machine learning algorithm. We only learn from problems that cannot be solved by the automatic mode within 60 seconds. The reason for this is that the learning time of the algorithm is cubic in the number of training examples which makes learning from all examples (the whole TPTP library) infeasible.

[2] Since the normalization function is defined during setup, the normalized features of new problems may fall out of the $[0, 1]$ interval.

[3] A possible improvement would be to take not only the predicted times, but also the orthogonality (difference in problems solved between strategies) of strategies into account. First experiments with this approach were promising.

Table 2. Results for the CASC@Turing problems

System	All Problems	New Problems
E 1.6	378/500	73/97
E-MaLeS 1.1	401/500	87/97

Table 3. Results for the CASC-J6 problems

System	All Problems	New Problems
E 1.6	359/450	50/68
E-MaLeS 1.1	377/450	59/68

A possible explanation of the discrepancy between old and new problems solved is that E 1.6 is overspecialized for the old problems. Cross-validation and regularization helps E-MaLeS to combat such overfitting.

E-MaLeS also competed in the LTB (Large Theory Batch) division of CASC-J6 and placed fourth with 83 problems solved after E 1.6 with 87 solved problems. The LTB division contains problems from large theories, namely problems exported from Isabelle, Mizar and SUMO. Unlike in the FOF division, the ATPs may use all cores of the machine. The LTB version of E-MaLeS uses the extra time and all available cores to run more strategies but is in no other way optimized. E 1.6, on the other hand, makes use of the batch structure to avoid repeated parsing of the large background theories, and also makes better use of the various SInE strategies that heuristically select relevant premises from the large theories.

4 Future Work

There are several ways to improve the current algorithm. Extracting features based on the FOF instead of the CNF representation of a formula could lead to better learning performance. Using different learning algorithms might allow us to run E-MaLeS without relying on E's automatic mode.

Strategies themselves are just combinations of many ATP parameters. An interesting application of machine learning to ATP is to develop new strategies by searching for such good parameter combinations systematically (by methods like hill-climbing) on a large corpus of problems. A related task is to produce a set of strategies that are highly orthogonal, i.e., that solve very different problems, so that their collective coverage is high.

We could also learn strategies based on conjecture features (like symbols) in consistently-named corpora like MPTP, instead of just using abstract features like on the TPTP problems.

The methods that we develop can probably be used for any ATP, and it would be interesting to see how such learning and optimization works for systems like Vampire [7], Z3 [6], and iProver [5]. Meta-systems like Isabelle/Sledgehammer [4,3] and MaLARea [15] could then use this deeper knowledge about ATPs to attack new conjectures with the strongest possible combinations of systems and strategies.

References

1. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.: Premise Selection for Mathematics by Corpus Analysis and Kernel Methods. CoRR abs/1108.3446 (2011); Accepted to JAR
2. Armando, A., Baumgartner, P., Dowek, G. (eds.): IJCAR 2008. LNCS (LNAI), vol. 5195. Springer, Heidelberg (2008)
3. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 116–130. Springer, Heidelberg (2011)
4. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic Proof and Disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 12–27. Springer, Heidelberg (2011)
5. Korovin, K.: iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). Armando et al. [2], 292–298
6. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Commun. 15(2-3), 91–110 (2002)
8. Schulz, S.: E - A Brainiac Theorem Prover. AI Commun. 15(2-3), 111–126 (2002)
9. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press, New York (2004)
10. Sörensson, N., Eén, N.: MiniSat 2.1 and MiniSat++ 1.0 SAT Race, Editions Technical Report, Chalmers University of Technology, Sweden (2008)
11. Stenz, G., Wolf, A.: E-SETHEO: An Automated^3 Theorem Prover – System Abstract. In: Dyckhoff, R. (ed.) TABLEAUX 2000. LNCS, vol. 1847, pp. 436–440. Springer, Heidelberg (2000)
12. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
13. Tammet, T.: Gandalf. Journal of Automated Reasoning 18, 199–204 (1997)
14. Urban, J.: MPTP 0.2: Design, Implementation, and Initial Experiments. J. Autom. Reasoning 37(1-2), 21–43 (2006)
15. Urban, J., Sutcliffe, G., Pudlák, P., Vyskocil, J.: MaLARea SG1- Machine Learner for Automated Reasoning with Semantic Guidanc. In: Armando et al. [2], pp. 441–456
16. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. J. Artif. Int. Res. 32(1), 565–606 (2008)

TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism

Jasmin Christian Blanchette[1] and Andrei Paskevich[2,3]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] LRI, Université Paris-Sud, CNRS, France
[3] INRIA Saclay – Île-de-France, France

Abstract. The TPTP World is a well-established infrastructure for automatic theorem provers. It defines several concrete syntaxes, notably an untyped first-order form (FOF) and a typed first-order form (TFF0), that have become de facto standards. This paper introduces the TFF1 format, an extension of TFF0 with rank-1 polymorphism. The format is designed to be easy to process by existing reasoning tools that support ML-style polymorphism. It opens the door to useful middleware, such as monomorphizers and other translation tools that encode polymorphism in FOF or TFF0. Ultimately, the hope is that TFF1 will be implemented in popular automatic theorem provers.

1 Introduction

The TPTP World [15] is a well-established infrastructure for supporting research, development, and deployment of automated reasoning tools. It owes its name to its vast problem library, the Thousands of Problems for Theorem Provers (TPTP) [14]. In addition, it specifies concrete syntaxes for problems and solutions: Dozens of reasoning tools implement the TPTP untyped clause normal form (CNF) and first-order form (FOF) for classical first-order logic with equality.

It has often been argued that the gap between the features supported by provers and those needed by applications is too wide, and that rich interchange formats are needed to address this disconnect [10, 18]. A growing number of reasoners can process the recently introduced TPTP "core" typed first-order form (TFF0) [17], with monomorphic types and interpreted arithmetic [9, 13], or the corresponding higher-order form (THF0) [2]. A polymorphic version of THF0, the full THF, is in the works [16].

Despite the variety of this offering, there is a strong desire in part of the automated reasoning community for a portable *polymorphic first-order format.* Many applications require polymorphism, notably interactive theorem provers and program specification languages; but lacking a suitable syntax, applications and provers must communicate via monomorphic formats. To make matters worse, there is no entirely satisfactory way to eliminate polymorphism: Monomorphization algorithms are necessarily incomplete, and it is difficult to encode polymorphism in a complete yet also sound and efficient manner, especially in the presence of interpreted types [3, 5, 11]. Tool authors are reduced to developing their own monomorphizers and type encodings, often using suboptimal schemes. Polymorphism arguably belongs in provers, where it can be implemented simply and efficiently, as demonstrated by Alt-Ergo [4].

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 414–420, 2013.

This paper describes the TFF1 format, an extension of TFF0 with rank-1 polymorphism. The extension was designed with the participation of members of the TPTP community, reflecting its needs. Besides compatibility with TFF0 and conceptual integrity with the upcoming full THF, an important design goal was to ensure that the format can easily be processed by existing reasoning tools that support ML-style polymorphism. TFF1 also opens the door to useful middleware, such as monomorphizers and other translation tools. The complete specification is available online.[1] The parts that TFF1 inherits from TFF0 are described in the TFF0 specification [17].

2 Syntax

Briefly, the types, terms, and formulas of TFF1 are analogous to those of TFF0, except that function and predicate symbols can be declared to be polymorphic, types can contain type variables, and n-ary type constructors are allowed. Type variables in type signatures and formulas are explicitly bound. Instances of polymorphic symbols are specified by explicit type arguments, rather than inferred.

Types. The *types* of TFF1 are built from *type variables* and *type constructors* of fixed arities. The usual conventions of TPTP apply: Type variables start with an uppercase letter and type constructors with a lowercase letter. The types `A`, `list(A)`, `list(bird)`, and `map(nat, list(B))` are all examples of well-formed types.

As in TFF0, the type `$i` of individuals is predefined but has no fixed semantics, whereas the arithmetic types `$int`, `$rat`, and `$real` are modeled by $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ [17]. It is perfectly acceptable for a TFF implementation to restrict itself to "pure TFF*k*," without arithmetic. TFF*k* with arithmetic is sometimes labeled "TFA*k*."

Type Signatures. Each function and predicate symbol occurring in a formula must be associated with a *type signature* that specifies the types of the arguments and, for functions, the result type. Type signatures can take any of the following forms:

(a) a type (predefined or user-defined);
(b) the Boolean pseudotype `$o` (the result "type" of predicate symbols);
(c) $(\tau_1 * \cdots * \tau_n) > \tilde{\tau}$ for $n > 0$, where $\tau_1, \ldots, \tau_n$ are types and $\tilde{\tau}$ is a type or `$o`;
(d) `!>[`α_1 `: $tType, ..., `α_n `: $tType]:` ς for $n > 0$, where $\alpha_1, \ldots, \alpha_n$ are distinct type variables and ς has one of the previous three forms.

In accordance with TFF0, the parentheses in form (c) are omitted if $n = 1$. The binder `!>` in form (d) denotes universal quantification. If ς is of form (c), it must be enclosed in parentheses. All type variables must be bound by a `!>`-binder.

Form (a) is used for monomorphic constants; form (b), for propositional constants, including the predefined symbols `$true` and `$false`; form (c), for monomorphic functions and predicates; and form (d), for polymorphic functions and predicates.

Type variables that are bound by `!>` without occurring in the type signature's body are called *phantom type variables.* These make it possible to specify operations and relations directly on types and provide a convenient way to encode type classes.

[1] `http://www21.in.tum.de/~blanchet/tff1spec.pdf`

Type Declarations. Type constructors can optionally be declared. The following declarations introduce a nullary type constructor `bird`, a unary type constructor `list`, and a binary type constructor `map`:

```
tff(bird, type, bird: $tType).
tff(list, type, list: $tType > $tType).
tff(map, type, map: ($tType * $tType) > $tType).
```

If a type constructor is used before being declared, its arity is determined by the first occurrence. Any later declaration must give it the same arity.

A declaration of a function or predicate symbol specifies its type signature. Every type variable occurring in a type signature must be bound by a `!>`-binder. The following declarations introduce a monomorphic constant `pi`, a polymorphic predicate `is_empty`, and a pair of polymorphic functions `cons` and `lookup`:

```
tff(pi, type, pi: $real).
tff(is_empty, type, is_empty : !>[A : $tType]: (list(A) > $o)).
tff(cons, type, cons : !>[A : $tType]: ((A * list(A)) > list(A))).
tff(lookup, type,
    lookup : !>[A : $tType, B : $tType]: ((map(A, B) * A) > B)).
```

If a function or predicate symbol is used before being declared, a default type signature is assumed: `($i * ⋯ * $i) > $i` for functions and `($i * ⋯ * $i) > $o` for predicates. If a symbol is declared after its first use, the declared signature must agree with the assumed signature. If a type constructor, function symbol, or predicate symbol is declared more than once, it must be given the same type signature up to renaming of bound type variables. All symbols share the same namespace.

Function and Predicate Application. To keep the required type inference to a minimum, every use of a polymorphic symbol must explicitly specify the type instance. A symbol with a type signature `!>[`α_1 `: $tType, ..., `α_m `: $tType]: ((`$\tau_1 * \cdots * \tau_n$`) > `$\tilde{\tau}$`)` must be applied to m type arguments and n term arguments. Given the above type signatures for `is_empty`, `cons`, and `lookup`, the term `lookup($int, list(A), M, 2)` and the atom `is_empty($i, cons($i, X, nil($i)))` are well-formed and contain free occurrences of the type variable `A` and the term variables `M` and `X`.

In keeping with TFF1's rank-1 polymorphic nature, type variables can only be instantiated with actual types. In particular, `$o`, `$tType`, and `!>`-binders cannot occur in type arguments of polymorphic symbols.

For systems that implement type inference, the following extension of TFF1 might be useful. When a type argument of a polymorphic symbol can be inferred automatically, it may be replaced with the wildcard `$_`. For example: `is_empty($_, cons($_, X, nil($_)))`. The producer of a TFF1 problem must be aware of the type inference algorithm implemented in the consumer to omit only redundant type arguments.

Type and Term Variables. Every variable in a TFF1 formula must be bound. The variable's type must be specified at binding time:

```
tff(bird_list_not_empty, axiom,
    ![B : bird, Bs : list(bird)]:
        ~ is_empty(bird, cons(bird, B, Bs))).
```

If the type and the preceding colon (`:`) are omitted, the variable is given type `$i`. Every type variable occurring in a TFF1 formula (whether in a type argument or in the type of a bound variable) must also be bound, with the pseudotype `$tType`:

```
tff(lookup_update_same, axiom,
    ![A : $tType, B : $tType, M : map(A, B), K : A, V : B]:
        lookup(A, B, update(A, B, M, K, V), K) = V).
```

A single quantifier cluster can bind both type and term variables. Universal and existential quantifiers over type variables are allowed under the propositional connectives, including equivalence, as well as under other quantifiers over type variables, but not in the scope of a quantifier over a term variable, to avoid dependent types.

Example. The following problem gives the general flavor of TFF1. It declares and axiomatizes `lookup` and `update` operations on maps and conjectures that `update` is idempotent for fixed keys and values.

```
tff(map, type, map : ($tType * $tType) > $tType).
tff(lookup, type,
    lookup : !>[A : $tType, B : $tType]: ((map(A, B) * A) > B)).
tff(update, type,
    update : !>[A : $tType, B : $tType]:
                 ((map(A, B) * A * B) > map(A, B))).
tff(lookup_update_same, axiom,
    ![A : $tType, B : $tType, M : map(A, B), K : A, V : B]:
        lookup(A, B, update(A, B, M, K, V), K) = V).
tff(lookup_update_diff, axiom,
    ![A : $tType, B : $tType, M : map(A, B), V : B, K : A, L : A]:
        (K != L => lookup(A, B, update(A, B, M, K, V), L) =
                   lookup(A, B, M, L))).
tff(map_ext, axiom,
    ![A : $tType, B : $tType, M : map(A, B), N : map(A, B)]:
        ((![K : A]: lookup(A, B, M, K) = lookup(A, B, N, K)) =>
         M = N)).
tff(update_idem, conjecture,
    ![A : $tType, B : $tType, M : map(A, B), K : A, V : B]:
        update(A, B, update(A, B, M, K, V), K, V) =
        update(A, B, M, K, V)).
```

3 Type Checking and Semantics

Notation. In this section, we use standard mathematical notation to write types, terms, and formulas. We use the symbols $\times$, $\to$, and $\forall$ to write type signatures, and write o for the Boolean pseudotype `$o`. It is convenient to treat $\approx$, $\neg$, $\wedge$, and $\forall$ as logical symbols and regard $\bot$, $\top$, $\not\approx$, $\vee$, $\to$, $\leftarrow$, $\leftrightarrow$, $\not\leftrightarrow$, and $\exists$ as abbreviations. Equality could be seen as a polymorphic predicate with the type signature $\forall\alpha.\ \alpha \times \alpha \to \mathrm{o}$, but the type instance is implicitly specified by the type of either argument, instead of explicitly via a type argument; hence, it is preferable to treat it as a logical symbol.

Type Checking. Let γ be a *type context*, a function that maps every variable to a type. A type judgment $\gamma \vdash t : \tau$ expresses that the term t is *well-typed* and has type τ in context γ. A type judgment $\gamma \vdash \varphi : \mathrm{o}$ expresses that the formula φ is *well-typed* in γ. We write $f : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \tau$ and $p : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \mathrm{o}$ to specify type signatures of function and predicate symbols, where m and n can be 0. The typing rules of TFF1 are as follows (where ρ is a type substitution):

$$\frac{}{\gamma \vdash u : \gamma(u)}$$

$$\frac{f : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \tau \qquad \gamma \vdash t_1 : \tau_1 \rho \quad \cdots \quad \gamma \vdash t_n : \tau_n \rho}{\gamma \vdash f(\alpha_1 \rho, \ldots, \alpha_m \rho, t_1, \ldots, t_n) : \tau \rho}$$

$$\frac{p : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \mathrm{o} \qquad \gamma \vdash t_1 : \tau_1 \rho \quad \cdots \quad \gamma \vdash t_n : \tau_n \rho}{\gamma \vdash p(\alpha_1 \rho, \ldots, \alpha_m \rho, t_1, \ldots, t_n) : \mathrm{o}}$$

$$\frac{\gamma \vdash s : \tau \qquad \gamma \vdash t : \tau}{\gamma \vdash s \approx t : \mathrm{o}} \qquad \frac{\gamma \vdash \varphi : \mathrm{o} \qquad \gamma \vdash \psi : \mathrm{o}}{\gamma \vdash \varphi \wedge \psi : \mathrm{o}}$$

$$\frac{\gamma \vdash \varphi : \mathrm{o}}{\gamma \vdash \neg \varphi : \mathrm{o}} \qquad \frac{\gamma[u \mapsto \tau] \vdash \varphi : \mathrm{o}}{\gamma \vdash \forall u : \tau.\, \varphi : \mathrm{o}} \qquad \frac{\gamma \vdash \varphi[\alpha'/\alpha] : \mathrm{o}}{\gamma \vdash \forall \alpha.\, \varphi : \mathrm{o}}$$

In the last rule, α' is an arbitrary type variable that occurs neither in φ nor in the values of γ. The renaming is necessary to reject formulas such as $\forall \alpha.\, \forall u : \alpha.\, \forall \alpha.\, \forall v : \alpha.\, u \approx v$, where the types of u and v are actually different. By assuming that no type variable can be both free and bound in the same formula, we can avoid explicit renaming of type variables, and the last typing rule's premise becomes $\gamma \vdash \varphi : \mathrm{o}$.

Semantics. An interpretation $\mathfrak{I}$ for a given set of type constructors, function symbols, and predicate symbols is constructed as follows. First, we fix a nonempty collection $\mathbb{D}$ of nonempty sets, the *domains.* The union of all domains is called the *universe*, $\mathbb{U}$.

An n-ary type constructor κ is interpreted as a function $\kappa^{\mathfrak{I}} : \mathbb{D}^n \to \mathbb{D}$. Let θ be a *type valuation*, a function that maps every type variable to a domain. Types are evaluated according to the equations $[\![\alpha]\!]^{\mathfrak{I}}_{\theta} \triangleq \theta(\alpha)$ and $[\![\kappa(\tau_1, \ldots, \tau_n)]\!]^{\mathfrak{I}}_{\theta} \triangleq \kappa^{\mathfrak{I}}\big([\![\tau_1]\!]^{\mathfrak{I}}_{\theta}, \ldots, [\![\tau_n]\!]^{\mathfrak{I}}_{\theta}\big)$. Since type evaluation depends only on the values of θ on the type variables occurring in a type, we write $[\![\tau]\!]^{\mathfrak{I}}$ to denote the domain of a ground type τ.

A predicate symbol $p : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \mathrm{o}$ is interpreted as a relation $p^{\mathfrak{I}} \subseteq \mathbb{D}^m \times \mathbb{U}^n$. A function symbol $f : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \tau$ is interpreted as a function $f^{\mathfrak{I}}$ on $\mathbb{D}^m \times \mathbb{U}^n$ that maps any m domains $D_1, \ldots, D_m$ and n universe elements to an element of $[\![\tau]\!]^{\mathfrak{I}}_{\theta}$, where θ maps each α_i to D_i.

Let ξ be a *variable valuation*, a function that assigns to every variable an element of $\mathbb{U}$. TFF1 terms and formulas are evaluated according to the following equations:

$$[\![u]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq \xi(u) \qquad [\![\neg \varphi]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq \neg\, [\![\varphi]\!]^{\mathfrak{I}}_{\theta,\xi}$$

$$[\![f(\bar{\sigma}, \bar{t})]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq f^{\mathfrak{I}}\big([\![\bar{\sigma}]\!]^{\mathfrak{I}}_{\theta}, [\![\bar{t}]\!]^{\mathfrak{I}}_{\theta,\xi}\big) \qquad [\![\varphi \wedge \psi]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq [\![\varphi]\!]^{\mathfrak{I}}_{\theta,\xi} \wedge [\![\psi]\!]^{\mathfrak{I}}_{\theta,\xi}$$

$$[\![p(\bar{\sigma}, \bar{t})]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq p^{\mathfrak{I}}\big([\![\bar{\sigma}]\!]^{\mathfrak{I}}_{\theta}, [\![\bar{t}]\!]^{\mathfrak{I}}_{\theta,\xi}\big) \qquad [\![\forall u : \tau.\, \varphi]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq \forall a \in [\![\tau]\!]^{\mathfrak{I}}_{\theta}.\, [\![\varphi]\!]^{\mathfrak{I}}_{\theta,\xi[u \mapsto a]}$$

$$[\![t_1 \approx t_2]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq \big([\![t_1]\!]^{\mathfrak{I}}_{\theta,\xi} = [\![t_2]\!]^{\mathfrak{I}}_{\theta,\xi}\big) \qquad [\![\forall \alpha.\, \varphi]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq \forall D \in \mathbb{D}.\, [\![\varphi]\!]^{\mathfrak{I}}_{\theta[\alpha \mapsto D],\xi}$$

4 Applications

A number of applications already support TFF1. Geoff Sutcliffe has extended the TPTP World infrastructure to process TFF1 problems and solutions. This involved adapting the Backus–Naur form specification of the TPTP syntaxes, from which parsers are generated.[2] Some TPTP tools still need to be ported to TFF1; this is ongoing work.

The Why3 [6] environment, which defines its own ML-like polymorphic specification language, can parse pure TFF1. Why3 translates between TFF1 and a wide range of formats, including FOF, SMT-LIB, and Alt-Ergo's native syntax [5, 7]. In addition, Why3's TFF1 parser is being ported to Alt-Ergo [4].

HOL(y)Hammer [8] and Sledgehammer [12] integrate various automatic provers in the proof assistants HOL Light and Isabelle/HOL. They have been extended to output pure TFF1 problems for Alt-Ergo and Why3. Using Sledgehammer, we produced 987 problems to populate the TPTP library.[3]

5 Conclusion

The TPTP TFF1 format complements the existing TPTP offerings. For reasoning tools that already support polymorphism, TFF1 is a portable alternative to the existing ad hoc syntaxes. But more importantly, the format is a vehicle to foster native polymorphism support in automatic reasoners. The time is ripe: After many years of untyped reasoning, we have recently witnessed the rise of interpreted arithmetic embedded in monomorphic logics. TFF1 lifts the most obvious restrictions of such systems.

The TPTP library already contains nearly a thousand TFF1 problems, and although the format is in its infancy, it is supported by several applications, including the SMT solver Alt-Ergo (via Why3). Work has commenced in Saarbrücken to add polymorphism to the superposition prover SPASS [19]. Given that many applications require polymorphism, other reasoning tools are likely to follow suit. The annual CADE Automated System Competition (CASC) will surely have a role to play driving adoption of the format. But regardless of progress in prover technology, equipped with a concrete syntax and suitable middleware, users can already turn their favorite automatic theorem prover into a fairly efficient polymorphic prover. Rank-1 polymorphism is, of course, no panacea; higher ranks and dependent types could be part of a future TFF2.

For SMT (satisfiability modulo theories) solvers, the SMT-LIB 2 format [1] specifies a classical many-sorted logic much in the style of TFF0 but with parametric symbol declarations (overloading). Polymorphism would make sense there as well, as witnessed by Alt-Ergo. However, the SMT community is still recovering from the upgrade to SMT-LIB 2 and busy defining a standard proof format; implementers would not welcome yet another feature at this point. Moreover, with its support for arithmetic, TFF1 is a reasonable format to implement in an SMT solver if polymorphism is desired.

Acknowledgment. The present specification is largely the result of consensus among participants of the `polymorphic-tptp-tff` mailing list, especially François Bobot, Chad Brown, Florian Rabe, Philipp Rümmer, Stephan Schulz, Geoff Sutcliffe, and Josef

[2] `http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html`

[3] `http://www.cs.miami.edu/~tptp/TPTP/Proposals/TFF1.html`

Urban. We are grateful to Geoff Sutcliffe, TPTP Master of Ceremonies, for giving TFF1 his benediction and adapting the TPTP BNF and other infrastructure. He, Mark Summerfield, and several anonymous reviewers suggested many textual improvements to this paper. We also thank Viktor Kuncak, Tobias Nipkow, Andrei Popescu, and Nicholas Smallbone for their support and ideas. The first author's research was supported by the Deutsche Forschungsgemeinschaft project Hardening the Hammer (grant Ni 491/14-1).

References

1. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard—Version 2.0. In: Gupta, A., Kroening, D. (eds.) SMT 2010 (2010)
2. Benzmüller, C., Rabe, F., Sutcliffe, G.: THF0 – the core of the TPTP language for higher-order logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 491–506. Springer, Heidelberg (2008)
3. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 493–507. Springer, Heidelberg (2013)
4. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: Barrett, C., de Moura, L. (eds.) SMT 2008, pp. 1–5. ICPS, ACM (2008)
5. Bobot, F., Paskevich, A.: Expressing polymorphic types in a many-sorted language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 87–102. Springer, Heidelberg (2011)
6. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Leino, K.R.M., Moskal, M. (eds.) Boogie 2011, pp. 53–64 (2011)
7. Couchot, J.F., Lescuyer, S.: Handling polymorphism in automated deduction. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 263–278. Springer, Heidelberg (2007)
8. Kaliszyk, C., Urban, J.: Learning-assisted automated reasoning with Flyspeck (submitted)
9. Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007)
10. Kuncak, V.: Intermediate languages—From birth to execution. In: Boogie 2011 (2011)
11. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010)
12. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Ternovska, E., Schulz, S. (eds.) IWIL 2010 (2010)
13. Prevosto, V., Waldmann, U.: SPASS+T. In: Sutcliffe, G., Schmidt, R., Schulz, S. (eds.) ESCoR 2006. CEUR Workshop Proceedings, vol. 192, pp. 18–33. CEUR-WS.org (2006)
14. Sutcliffe, G.: The TPTP problem library and associated infrastructure—The FOF and CNF parts, v3.5.0. J. Autom. Reasoning 43(4), 337–362 (2009)
15. Sutcliffe, G.: The TPTP world – infrastructure for automated reasoning. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 1–12. Springer, Heidelberg (2010)
16. Sutcliffe, G., Benzmüller, C.: Automated reasoning in higher-order logic using the TPTP THF infrastructure. J. Formal. Reasoning 3(1), 1–27 (2010)
17. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP typed first-order form with arithmetic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 406–419. Springer, Heidelberg (2012)
18. Voronkov, A.: Automated reasoning: Past story and new trends. In: Gottlob, G., Walsh, T. (eds.) IJCAI 2003, pp. 1607–1612. Morgan Kaufmann (2003)
19. Wand, D., Weidenbach, C.: Private communication (June 2012)

Propositional Temporal Proving with Reductions to a SAT Problem

Richard Williams and Boris Konev

Department of Computer Science, University of Liverpool, Liverpool L69 7ZF
{R.M.Williams1,Konev}@liv.ac.uk

Abstract. We present a new approach to reasoning in propositional linear-time temporal logic (PLTL). The method is based on the simplified temporal resolution calculus. We prove that the search for premises to apply the rules of simplified temporal resolution can be re-formulated as a search for minimal unsatisfiable subsets (MUS) in a set of classical propositional clauses. This reformulation reduces a large proportion of PLTL reasoning to classical propositional logic facilitating the use of modern tools. We describe an implementation of the method using the CAMUS system for MUS computation and present an in-depth comparison of the performance of the new solver against a clausal temporal resolution prover.

1 Introduction

Propositional Linear-time Temporal Logic (PLTL) is an extension of classical propositional logic with operators that deal with time. PLTL, and its extensions, have been used in various areas of computer science, for example, for the specification of distributed and concurrent systems and verification of their properties through temporal reasoning [20], for synthesis of programs from temporal specifications [23], in temporal databases [27] and for knowledge representation and reasoning [15]. PLTL is notable for its widespread use as a specification language in software and hardware verification via model checking [6].

In recent years, we witness a renewed interest in PLTL theorem proving. Among other reasons, it can be explained by the fact that PLTL specifications, used in verification of software and hardware systems, often go far beyond simple safety and liveness conditions. In fact, temporal specifications became so complicated that a need arises for an automated check if they are (un)satisfiable. Indeed, it does not make sense to check whether a PLTL formula is true in a model if the formula is unsatisfiable or valid [12,25,24].

Satisfiability of PLTL formulae can be established with a number of techniques including automata-based approaches [28], tableau methods [30] and clausal temporal resolution [11]. Clausal temporal resolution has been successfully implemented [18,17] and shown to perform well in practice [17,25].

Clausal temporal resolution is a machine-oriented calculus that operates on temporal formulae in a clausal form called SNF and uses a small number of resolution rules. In a nutshell, clausal temporal resolution propagates conflicts

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 421–435, 2013.

'backward in time' until a contradiction is derived in the initial state [11]. For example, consider a conjunction of the following three temporal clauses

$$1\colon a \Rightarrow \bigcirc(x \vee y) \quad 2\colon b \Rightarrow \bigcirc\neg x \quad 3\colon c \Rightarrow \bigcirc\neg y,$$

where $\bigcirc$ denotes 'at the next moment of time'. Similar to classical resolution, clausal temporal resolution derives a new clause $4\colon a \wedge b \Rightarrow \bigcirc y$ [`from 1 and 2`]. Then the derived clause 4 can be combined with clause 3 to derive $5\colon a \wedge b \wedge c \Rightarrow \bigcirc\mathbf{false}$ [`from 4 and 3`]. This latter clause can be rewritten as $6\colon \neg a \vee \neg b \vee \neg c$ [`from 5`]. Indeed, if it is not the case that at least one of a, b or c is false, we inevitably get a contradiction at the next moment of time.

Simplified temporal resolution introduced in [7] derives clause 6 in one go by noticing that the (pure classical, that is, containing no temporal operators) conjunction of the right-hand sides of the given clauses, $(x \vee y) \wedge \neg x \wedge \neg y$ is unsatisfiable. Thus, an application of the temporal resolution rule can be characterised in an abstract way as a multi-premise rule with a purely classical side condition.

The biggest challenge in implementing the simplified calculus is that the abstract characterisation of the inference rules gives no hint on which temporal clauses need to be combined. A straightforward implementation of simplified temporal resolution enumerates all combinations of temporal clauses in order to find those satisfying the classical side conditions. The resulting procedure is best-case exponential. In fact, simplified temporal resolution was never intended to be implemented. The calculus has primarily been introduced to provide a cleaner separation between temporal and classical reasoning, to simplify the proof of completeness and to explore variations of the clausal normal form [7].

In this paper we present a new approach to PLTL reasoning based on simplified temporal resolution, which tackles the challenge of determining which clauses need to be combined by reducing it to the propositional Minimal Unsatisfiable Subset (MUS) problem. A set of propositional clauses is an MUS if it is both usatisfiable and any proper subset is satisfiable. We prove that when searching for temporal clauses to combine for simplified temporal reasoning, it suffices to consider those whose right-hand side (together with some universal clauses) forms an MUS. This reduces a large proportion of PLTL reasoning to classical propositional logic. We report on a rigorous experimental evaluation of our prototype implementation of the calculus, which shows that this simple and elegant idea works well in practice.

2 Preliminaries

The set of PLTL formulae is the smallest set containing the set of (atomic) propositions *Prop* and such that if ϕ and ψ are in PLTL formulae, then so are **true**, $\neg\phi$, $\phi \vee \psi$, $\bigcirc\phi$ ('ϕ is true in the next moment'), and $\phi \mathsf{U} \psi$ ('ϕ is true until ψ becomes true'). As usual, we introduce other Boolean and temporal operators ($\Box$ 'always in the future', $\Diamond$ 'sometime in the future' and W 'unless')

as abbreviations: $\mathbf{false} = \neg\mathbf{true}$, $\phi \wedge \psi = \neg(\neg\phi \vee \neg\psi)$, $\phi \Rightarrow \psi = \neg\phi \vee \psi$, $\phi \Leftrightarrow \psi = (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$, $\Diamond\phi = \mathbf{true}\ \mathsf{U}\ \phi$, $\Box\phi = \neg\Diamond\neg\phi$ and $\phi\ \mathsf{W}\ \psi = (\phi\ \mathsf{U}\ \psi) \vee (\Box\phi)$.

A PLTL formula that does not contain any temporal operators is called a (classical) *propositional formula*. A *literal* is a proposition or a negation of a proposition. A *clause* is disjunction of literals. A propositional CNF formula is a conjunction of clauses. We do not make a distinction between a propositional CNF formula ϕ and the set of clauses S such that $\phi = \bigwedge_{C \in S} C$.

A model for a PLTL formula ϕ can be characterised as a sequence of *states* of the form $\sigma = s_0, s_1, s_2, \ldots$, where each state s_i is a set of propositions that are satisfied at the i^{th} moment in time. We call every such sequence of states an interpretation. We define the relation $(\sigma, i) \models \phi$ (at time instance i, interpretation σ satisfies PLTL formula ϕ) by induction on the structure of the formula as follows:

$$\begin{array}{lll}
(\sigma,i) \models p & \text{iff} & p \in s_i \quad [\text{for } p \in \mathit{Prop} \text{ and } \sigma = s_0, s_1, \ldots] \\
(\sigma,i) \models \bigcirc\psi & \text{iff} & (\sigma, i+1) \models \psi \\
(\sigma,i) \models \phi\ \mathsf{U}\ \psi & \text{iff} & \text{iff } \exists k \in \mathbb{N}.\ k \geq i \text{ and } (\sigma,k) \models \psi \text{ and} \\
 & & \forall j \in \mathbb{N}, \text{ if } i \leq j < k \text{ then } (\sigma,j) \models \phi
\end{array}$$

We say that a formula ϕ is *satisfiable* if, and only if, there exists an interpretation σ such that $(\sigma, 0) \models \phi$. We also say that σ is a model of ϕ in this case. A formula ϕ is *valid* if, and only if, it is satisfied in every possible interpretation, i.e. for each σ, $(\sigma, 0) \models \phi$. A formula ϕ is unsatisfiable if, and only if, it is not satisfiable.

Simplified temporal resolution introduced in [7] operates on temporal problems in divided separated normal form (DSNF). A DSNF problem is a quadruple $\langle \mathcal{I}, \mathcal{U}, \mathcal{S}, \mathcal{E} \rangle$, where

- $\mathcal{I}$ (the initial part) and $\mathcal{U}$ (the universal part) are sets of propositional clauses;
- $\mathcal{S}$ (the step part) is a set of *step clauses* of the form

$$P \Rightarrow \bigcirc Q,$$

  where P is a conjunction of literals and Q is a disjunction of literals;
- and $\mathcal{E}$ (the eventuality part) is a set of *eventualities* of the form $\Diamond l$, where l is a literal.

The intended meaning of a DSNF problem is given by

$$\mathcal{I} \wedge \Box\mathcal{U} \wedge \Box\mathcal{S} \wedge \Box\mathcal{E}.$$

When we talk about particular properties of temporal problems (e.g., satisfiability, validity, logical consequences etc) we mean properties of the associated formula. (As above, we do not make a distinction between a finite set of formulae and a conjunction of formulae in this set.)

Arbitrary PLTL-formulae can be transformed into satisfiability equivalent DSNF problems using a renaming technique replacing non-atomic subformulae with new propositions and replacing all occurrences of the U ('until'), $\Box$

('always') and W ('unless') operators with their fixpoint definitions. The size of the resulting temporal problem in DSNF is at most linear in the size of the given formula [11,7,8]. We illustrate DSNF transformation with an example.

Example 1. Consider temporal formula $\phi = \Diamond\Box a \wedge \Diamond\Box\neg a$. First, we satisfiability equivalently rewrite it as $x_1 \wedge \Box(x_1 \Rightarrow \Diamond\Box a) \wedge \Box(x_1 \Rightarrow \Diamond\Box\neg a)$, where x_1 a fresh proposition. Then we rename the occurrences of the always operator and then 'unwind' the always operator using its fixpoint definition to give

$$\begin{array}{c} x_1 \wedge \Box(x_1 \Rightarrow \Diamond x_2) \wedge \Box(x_1 \Rightarrow \Diamond x_3) \wedge \\ \Box(x_2 \Rightarrow \bigcirc x_2) \wedge \Box(x_2 \Rightarrow a) \wedge \\ \Box(x_3 \Rightarrow \bigcirc x_3) \wedge \Box(x_3 \Rightarrow \neg a), \end{array}$$

where x_2 and x_3 are fresh propositions. Then we replace conditional eventualities $\Box(x_1 \Rightarrow \Diamond x_2)$ and $\Box(x_1 \Rightarrow \Diamond x_3)$ with unconditional ones. Formula $\Box(x_1 \Rightarrow \Diamond x_2)$ is satisfiability equivalent to $\Box(x_1 \Rightarrow (x_2 \vee w_1)) \wedge \Box(w_1 \Rightarrow \bigcirc(w_1 \vee x_2)) \wedge \Box\Diamond\neg w_1$, where w_1 is a fresh proposition, which, intuitively, is true 'while we are waiting for x_2 to become true'; the other eventuality is treated similarly. All in all, ϕ is satisfiability equivalent to the following temporal DSNF problem.

$$\mathcal{I} = \{\mathsf{i1}\colon x_1\}; \qquad \mathcal{U} = \left\{ \begin{array}{l} \mathsf{u1}\colon \neg x_2 \vee a, \\ \mathsf{u2}\colon \neg x_3 \vee \neg a, \\ \mathsf{u3}\colon \neg x_1 \vee x_2 \vee w_1, \\ \mathsf{u4}\colon \neg x_1 \vee x_3 \vee w_2 \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{l} \mathsf{s1}\colon x_2 \Rightarrow \bigcirc x_2, \\ \mathsf{s2}\colon x_3 \Rightarrow \bigcirc x_3, \\ \mathsf{s3}\colon w_1 \Rightarrow \bigcirc(w_1 \vee x_2), \\ \mathsf{s4}\colon w_2 \Rightarrow \bigcirc(w_2 \vee x_3) \end{array} \right\}; \mathcal{E} = \{\mathsf{e1}\colon \Diamond\neg w_1, \mathsf{e2}\colon \Diamond\neg w_2\}.$$

The added labels i1, u1,... have no special meaning and are not part of DSNF; we use them for reference when we return to this example. □

Simplified temporal resolution consists of an (implicit) *merging operation*

$$\frac{P_1 \Rightarrow \bigcirc Q_1, \ldots, P_n \Rightarrow \bigcirc Q_n}{\bigwedge\limits_{j=1}^{n} P_i \Rightarrow \bigcirc \bigwedge\limits_{j=1}^{n} Q_i},$$

and resolution and termination rules defined below. To simplify the presentation, we denote the result of merging of step clauses as $\mathcal{A} \Rightarrow \bigcirc\mathcal{B}$ (or $\mathcal{A}_i \Rightarrow \bigcirc\mathcal{B}_i$ if a rule operates several merged step clauses). Thus, in what follows $\mathcal{A}$, $\mathcal{B}$, $\mathcal{A}_i$ and $\mathcal{B}_i$ are conjunctions of propositional literals. As $\mathcal{U}$ contains no temporal operators, all side conditions in the rules are purely propositional.

– *Step resolution rule*:

$$\frac{\mathcal{A} \Rightarrow \bigcirc\mathcal{B}}{\neg\mathcal{A}},$$

where $\mathcal{U} \cup \{\mathcal{B}\}$ is unsatisfiable.

function SRes($\mathcal{U}$, $\mathcal{S}$)
  $New = \emptyset$
  for each $mus \in$ AllMus $(\{B \mid A \Rightarrow \bigcirc B \in \mathcal{S}\} \cup \mathcal{U})$ **do**
    $\mathcal{A} = \bigwedge_{B \in mus,\, A \Rightarrow \bigcirc B \in \mathcal{S}} A$
    $New = New \cup \{\neg \mathcal{A}\}$
  end for
  return New
end function

Fig. 1. Step resolution

– *Eventuality resolution rule*:

$$\frac{\mathcal{A}_1 \Rightarrow \bigcirc \mathcal{B}_1, \quad \ldots, \quad \mathcal{A}_n \Rightarrow \bigcirc \mathcal{B}_n \qquad \Diamond l}{(\bigwedge_{i=1}^{n} \neg \mathcal{A}_i)} ,$$

  where $\mathcal{U} \cup \{\mathcal{B}_i, l\}$ and $\mathcal{U} \cup \{\mathcal{B}_i, \bigwedge_{j=1}^{n} \neg \mathcal{A}_j\}$, for all i, are unsatisfiable.
– *Termination rule*: **false** is derived if $\mathcal{U} \cup \mathcal{I}$, or $\mathcal{U} \cup \{l\}$ are unsatisfiable, where l is an eventuality literal.

A *derivation* is a sequence of universal parts, $\mathcal{U} = \mathcal{U}_0 \subset \mathcal{U}_1 \subset \mathcal{U}_2 \subset \ldots$, extended little by little by the conclusions of the inference rules. Notice that, as the left-hand sides of (merged) step clauses are conjunctions of literals, the step resolution rule generates clauses and the eventuality resolution rule generates sets of clauses. The $\mathcal{I}$, $\mathcal{S}$ and $\mathcal{E}$ parts of the temporal problem are not changed during a derivation. A derivation *terminates* if, and only if, either **false** is derived, in which case we say that the derivation *successfully terminates*, or if no new formulae can be derived by further inference steps. A derivation $\mathcal{U} = \mathcal{U}_0 \subset \mathcal{U}_1 \subset \mathcal{U}_2 \subset \cdots \subset \mathcal{U}_n$ is called *fair* if for any $i \geq 0$ and formula ϕ derivable from $\langle \mathcal{U}_i, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ by the rules above, there exists $j \geq i$ such that $\phi \in \mathcal{U}_j$.

Theorem 1 ([7]). *If a DSNF problem $\langle \mathcal{I}, \mathcal{U}, \mathcal{S}, \mathcal{E} \rangle$ is unsatisfiable then any fair derivation by temporal resolution successfully terminates.*

3 Temporal Reasoning with Reductions to MUS

As the side conditions of the inference rules are purely propositional problems, they can be tested with an external SAT Solver. All that remains is to find the appropriate merged step clause, or clauses, which satisfy the side conditions. This straightforward approach has been implemented in [29]; however, in practice, the necessity to try all possibilities to merge clauses led to inability to handle problems with a sizeable step part. In this paper we investigate a possibility to delegate the search for step clauses to merge to an MUS solver.

For an unsatisfiable set of propositional clauses S, its subset $S' \subseteq S$ is called a *minimal unsatisfiable subset* (MUS) if S' is unsatisfiable and every proper

function ERes($\mathcal{U}$, $\mathcal{S}$, $\Diamond l$)
  $H = \text{SRes}(\mathcal{U} \cup \{l\}, \mathcal{S})$
  repeat
    $H' = H$; $H = \text{SRes}(\mathcal{U} \cup \{(l \vee \neg\mathcal{A}') \mid \neg\mathcal{A}' \in H'\}, \mathcal{S})$
    if $H = \emptyset$ **then**
      return $\emptyset$
    end if
  until $(\bigwedge_{\neg\mathcal{A}\in H} \neg\mathcal{A} \Rightarrow \bigwedge_{\neg\mathcal{A}'\in H'} \neg\mathcal{A}')$ is valid
  return H
end function

Fig. 2. Eventuality resolution

subset of S' is satisfiable. The number of MUSes for a set of clauses S can be exponential in the size of S. Propositional minimal unsatisfiability has been extensively studied (often under different names) in the literature, and a number of empirically efficient implementations of algorithms enumerating all MUSes for a given set of propositional clauses is available (see the survey [21] and references within).

The step resolution procedure is given in Figure 1. The AllMus procedure called returns all MUSes for a set of propositional clauses. By definition, every $\neg\mathcal{A} \in \text{SRes}(\mathcal{S},\mathcal{U})$ is obtained from $\langle\mathcal{I},\mathcal{U},\mathcal{S},\mathcal{E}\rangle$ by an application of the step resolution rule; conversely we have the following.

Lemma 1. *For any DNSF problem $\mathcal{P} = \langle\mathcal{I},\mathcal{U},\mathcal{S},\mathcal{E}\rangle$ such that $\mathcal{U}$ is satisfiable, if $\neg\mathcal{A}$ can be obtained by an application of the step resolution rule from $\mathcal{P}$, then there exists $\neg\mathcal{A}' \in \mathsf{SRes}(\mathcal{S},\mathcal{U})$ such that $(\neg\mathcal{A}' \Rightarrow \neg\mathcal{A})$ is a valid formula.*

Proof. Let $\mathcal{B} = \bigwedge_{i\in I} B_i$. As $\mathcal{U}$ is satisfiable and $\mathcal{B} \wedge \mathcal{U}$ is not, there exists $J \subseteq I$ such that for some MUS mus we have $B_j \in mus$, for every $j \in J$, so $\neg(\bigwedge_{j\in J} A_j)$ will be returned by $\text{SRes}(\mathcal{U},\mathcal{S})$. Clearly, $(\neg(\bigwedge_{j\in J} A_j) \Rightarrow \neg\mathcal{A})$ is valid. □

It has been noticed already in [9] that the search for premises of the eventuality resolution rule can be performed with the help of step resolution. Our algorithm for eventuality resolution given in Figure 2 is based on the BFS algorithm as described in [8]. Notice that every element of the set H in the $\text{ERes}(\mathcal{U}, \mathcal{S}, \Diamond l)$ procedure is of the form $\neg\mathcal{A}$, where $\mathcal{A}$ is the left-hand side of some merged step clause $\mathcal{A} \Rightarrow \bigcirc\mathcal{B}$.

We demonstrate the working of the ERes procedure by proving its correctness; the proof of completeness of simplified temporal resolution with the eventuality rule applications restricted to the outputs of ERes can be obtained by adapting the proof of completeness in [8] using arguments similar to those used in the proof of Lemma 1.

Lemma 2. *For any DNSF problem $\mathcal{P} = \langle\mathcal{I},\mathcal{U},\mathcal{S},\mathcal{E}\rangle$ let H be returned by $\text{ERes}(\mathcal{U},\mathcal{S},\Diamond l)$. Then H can be obtained from $\langle\mathcal{I},\mathcal{U},\mathcal{S},\mathcal{E}\rangle$ by an application of the eventuality resolution rule.*

```
function PLTL(I,U,S,E)
    repeat
        if (U = U ∪ SRes(U,S) changes U) then
            check for termination
        else if (U = U ∪ ERes(U,S,◇l), for some ◇l ∈ E, changes U) then
            check for termination
        end if
    until There is no change in U
    check for termination
    return 'satisfiable'
end function
```

Fig. 3. Reasoning procedure

Proof. Suppose that $\mathrm{ERes}(\mathcal{U},\mathcal{S},\Diamond l)$ returns a non-empty set of clauses H and let H' be from the last iteration of the loop. Let a set of indices I be such that $H = \{\neg\mathcal{A}_i \mid i \in I\}$ and let $\mathcal{B}_i$ be such that $\mathcal{A}_i \Rightarrow \bigcirc\mathcal{B}_i$ is a merged step clause, for $i \in I$. Then, by properties of $\mathrm{SRes}(\mathcal{U},\mathcal{S})$, for every $i \in I$ the set $\{\mathcal{B}_i\} \cup \{(l \vee \neg\mathcal{A}') \mid \neg\mathcal{A}' \in H'\} \cup \mathcal{U}$ is unsatisfiable. As $(\bigwedge_{\neg\mathcal{A}\in H} \neg\mathcal{A} \Rightarrow \bigwedge_{\neg\mathcal{A}'\in H'} \neg\mathcal{A}')$ is valid, the set $\{\mathcal{B}_i\} \cup \{(l \vee \neg\mathcal{A}) \mid \neg\mathcal{A} \in H\} \cup \mathcal{U}$ is also unsatisfiable. Equivalently, $\mathcal{U} \cup \{\mathcal{B}_i, l\}$ and $\mathcal{U} \cup \{\mathcal{B}_i, \bigwedge_{j\in I} \neg\mathcal{A}_j\}$, for all $i \in I$, are unsatisfiable. But then H can be obtained by an application of the eventuality resolution rule. □

Finally, the overall proof procedure is given in Figure 3. Note in the procedure *check for termination* stands for checking if $\mathcal{U} \cup \mathcal{I}$ or $\mathcal{U} \cup \{l\}$ (for some $\Diamond l \in \mathcal{E}$) are unsatisfiable, in which case proof search terminates returning '*unsatisfiable*'. Combining Lemmata 1 and 2 with results from [8] we obtain the following result.

Theorem 2. $\mathrm{PLTL}(\mathcal{I},\mathcal{U},\mathcal{S},\mathcal{E})$ *always terminates. DSNF problem* $\langle\mathcal{I},\mathcal{U},\mathcal{S},\mathcal{E}\rangle$ *is satisfiable if, and only if,* $\mathrm{PLTL}(\mathcal{I},\mathcal{U},\mathcal{S},\mathcal{E})$ *returns 'satisfiable'.*

Example 2 (Example 1 continued). We apply our algorithm to the DSNF problem from Example 1. To simplify the notation, we refer to clauses just by their label. Additionally, we refer to the propositional clause in the right-hand side of a step clause by adding the suffix 'r' to the label of the step clause. For example, s3r denotes $w_1 \vee x_2$, the right-hand side of step clause s3.

When the algorithms starts $\mathcal{U} = \{\mathsf{u1}, \mathsf{u2}, \mathsf{u3}, \mathsf{u4}\}$. We apply PLTL step by step.

SRes($\mathcal{U}$, $\mathcal{S}$). AllMus($\{\mathsf{u1}, \mathsf{u2}, \mathsf{u3}, \mathsf{u4}, \mathsf{s1r}, \mathsf{s2r}, \mathsf{s3r}, \mathsf{s4r}\}$), returns just one MUS $\{\mathsf{u1}, \mathsf{u2}, \mathsf{s1r}, \mathsf{s2r}\}$. As $\{\mathsf{s1r}, \mathsf{s2r}\} \cup \mathcal{U}$ is unsatisfiable, the step resolution rule applies to the result of merging of s1 and s2 generating new universal clause: u5: $\neg x_2 \vee \neg x_3$, which is returned by the procedure and added to $\mathcal{U}$. As $\mathcal{U}$ is changed, the *check for termination* is employed, but it does not succeed. Then $\mathrm{SRes}(\mathcal{U}, \mathcal{S})$ is called again, but one can see that the second call leads to no change in $\mathcal{U}$.

ERes$(\mathcal{U}, \mathcal{S}, \Diamond\neg w_1)$**.** In order to compute H, SRes$(\mathcal{U} \cup \{\mathsf{t1}\colon \neg w_1\}, \mathcal{S})$ is called, where $\mathsf{t1}$ is a label used to denoted the temporary universal clause. AllMus$(\{\mathsf{u1}, \mathsf{u2}, \mathsf{u3}, \mathsf{u4}, \mathsf{u5}, \mathsf{t1}, \mathsf{s1r}, \mathsf{s2r}, \mathsf{s3r}, \mathsf{s4r}\})$ returns a set containing 4 MUSes, $\{\{\mathsf{s1r}, \mathsf{s2r}, \mathsf{u1}, \mathsf{u2}\}, \{\mathsf{s1r}, \mathsf{s2r}, \mathsf{u5}\}, \{\mathsf{s2r}, \mathsf{s3r}, \mathsf{t1}, \mathsf{u1}, \mathsf{u2}\}, \{\mathsf{s2r}, \mathsf{s3r}, \mathsf{t1}, \mathsf{u5}\}\}$ however H only contains 2 clauses $\{\neg x_2 \vee \neg x_3, \neg x_3 \vee \neg w_1\}$ as the two first and the two last MUSes contain the same right-hand sides of step clauses. We set $H' = H$ and run the loop body. The call of SRes$(\mathcal{U} \cup \{\mathsf{t2}\colon \neg w_1 \vee \neg x_2 \vee \neg x_3, \mathsf{t3}\colon \neg w_1 \vee \neg x_3\}, \mathcal{S})$ passes $\{\mathsf{u1}, \mathsf{u2}, \mathsf{u3}, \mathsf{u4}, \mathsf{u5}, \mathsf{t2}, \mathsf{t3}, \mathsf{s1r}, \mathsf{s2r}, \mathsf{s3r}, \mathsf{s4r}\}$ to AllMus which returns $\{\{\mathsf{s1r}, \mathsf{s2r}, \mathsf{u1}, \mathsf{u2}\}, \{\mathsf{s1r}, \mathsf{s2r}, \mathsf{u5}\}, \{\mathsf{s2r}, \mathsf{s3r}, \mathsf{t3}, \mathsf{u1}, \mathsf{u2}\}, \{\mathsf{s2r}, \mathsf{s3r}, \mathsf{t3}, \mathsf{u5}\}\}$, so $H = \{\neg x_2 \vee \neg x_3, \neg x_3 \vee \neg w_1\}$. As $H = H'$, ERes terminates and returns two universal clauses $\mathsf{u6}\colon \neg x_2 \vee \neg x_3$ and $\mathsf{u7}\colon \neg x_3 \vee \neg w_1$. Only $\mathsf{u7}$ is new and is added to $\mathcal{U}$; $\mathsf{u6}$ is discarded as redundant. As $\mathcal{U}$ is changed, the *check for termination* is employed, but it does not succeed.

ERes$(\mathcal{U}, \mathcal{S}, \Diamond\neg w_2)$**.** H is computed. SRes$(\mathcal{U} \cup \{\mathsf{t5}\colon \neg w_2\}, \mathcal{S})$ is called and AllMus$(\{\mathsf{u1}, \mathsf{u2}, \mathsf{u3}, \mathsf{u4}, \mathsf{u5}, \mathsf{t5}, \mathsf{s1r}, \mathsf{s2r}, \mathsf{s3r}, \mathsf{s4r}\})$ returns $\{\{\mathsf{s3r}, \mathsf{s4r}, \mathsf{u1}, \mathsf{u2}, \mathsf{u7}, \mathsf{t5}\}, \{\mathsf{s3r}, \mathsf{s4r}, \mathsf{u7}, \mathsf{t5}, \mathsf{u5}\}, \{\mathsf{s2r}, \mathsf{s3r}, \mathsf{u1}, \mathsf{u2}, \mathsf{u7}\}, \{\mathsf{s2r}, \mathsf{s3r}, \mathsf{u7}, \mathsf{u5}\}, \{\mathsf{s1r}, \mathsf{s4r}, \mathsf{u1}, \mathsf{u2}, \mathsf{t5}\}, \{\mathsf{s1r}, \mathsf{s4r}, \mathsf{t5}, \mathsf{u5}\}, \{\mathsf{s1r}, \mathsf{s2r}, \mathsf{u1}, \mathsf{u2}\}, \{\mathsf{s1r}, \mathsf{s2r}, \mathsf{u5}\}\}$ so $H = \{\neg x_2 \vee \neg x_3, \neg x_3 \vee \neg w_1, \neg x_2 \vee \neg w_2, \neg w_1 \vee \neg w_2\}$. The run of the loop body is omitted to save space, however it computes H being same as H', so ERes terminates and returns four universal clauses $\{\mathsf{u8}\colon \neg x_2 \vee \neg x_3, \mathsf{u9}\colon \neg x_3 \vee \neg w_1, \mathsf{u10}\colon \neg x_2 \vee \neg w_2, \mathsf{u11}\colon \neg w_1 \vee \neg w_2\}$. Only $\mathsf{u10}$ and $\mathsf{u11}$ are new, which are added to $\mathcal{U}$.

Finally as $\mathcal{U} \cup \mathcal{I}$ is unsatisfiable we can apply the termination rule and so PLTL$(\mathcal{I}, \mathcal{U}, \mathcal{S}, \mathcal{E})$ returns 'unsatisfiable'. □

Optimisations. As Example 2 demonstrates, different MUSes can contain the same right-hand sides of step clauses, which is not optimal. The search for merged clauses can be significantly sped up by *grouping* universal clauses together so that instead of looking for all minimal unsatisfiable subsets of $\{B \mid A \Rightarrow \bigcirc B \in \mathcal{S}\} \cup \mathcal{U}$, we look for all subsets $S \subseteq \{B \mid A \Rightarrow \bigcirc B \in \mathcal{S}\}$ such that $(S \cup \mathcal{U})$ is unsatisfiable and for every proper subset S' of S, $(S' \cup \mathcal{U})$ is satisfiable. In other words, all universal clauses are considered as one item. Not only is the number of MUSes with grouped universal clauses smaller than the number of all MUSes but also, crucially, MUS enumeration tools can efficiently take grouping into account [19].

We further exploit the disparity between the treatment of right-hand sides of step clauses and universal clauses by rewriting a given DNSF problem into a satisfiability equivalent problem having a smaller number of step clauses. If $\mathcal{S}$ contains two step clauses $A \Rightarrow \bigcirc B_1$ and $A \Rightarrow \bigcirc B_2$ with the same left-hand side, we first equivalently rewrite them into $A \Rightarrow \bigcirc(B_1 \wedge B_2)$ and then rename the conjunction $B_1 \wedge B_2$, to preserve the clausal form, to give a new step clause $A \Rightarrow \bigcirc X$ and two new universal clauses $\neg X \vee B_1$ and $\neg X \vee B_2$, where X is a fresh proposition. Similarly, if $\mathcal{S}$ contains two step clauses $A_1 \Rightarrow \bigcirc B$ and $A_2 \Rightarrow \bigcirc B$ with the same right-hand side, we equivalently rewrite them into $(A_1 \vee A_2) \Rightarrow \bigcirc B$ and then rename the disjunction in the left-hand side.

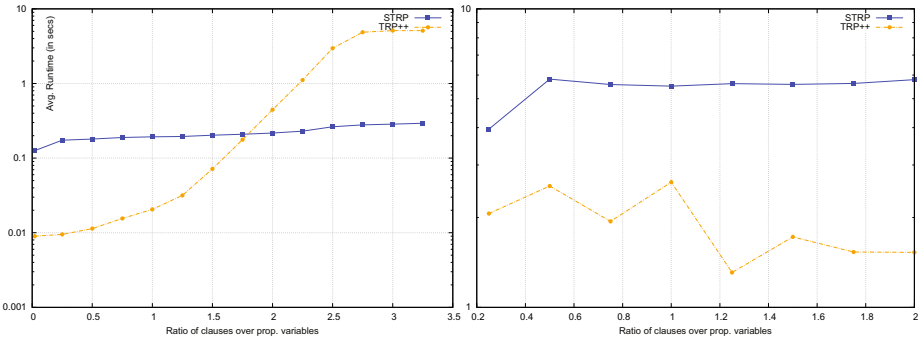


Fig. 4. Results for C^1_{ran} (left) and C^2_{ran} (right)

4 Experimental Evaluation

We have implemented the described approach in the STRP prover[1], which uses the CAMUS system [19] as the MUS enumeration tool. CAMUS works in two stages, the first extracts all minimal correction subsets (MCSes) from the input propositional clauses and the second extracts MUSes from the set of MCSes by extracting all minimal hitting sets (also know as minimal hypergraph transversals) from the set of all MCSes. Although both stages are not tractable [19], in our preliminary experiments, the second stage of CAMUS took much more time than the first stage. We put this down to the nature of our problems: SAT benchmarks typically are larger problems with a smaller number of MUSes [1], whereas our MUS problems are much smaller but typically contain a larger number of MUSes. From a small, randomly selected, sample of the benchmarks used in this work we have established a typical CNF problem size of 800–900 variables and 3500–4500 clauses (of which 120–130 were the right-hand sides of step clauses), from which about 1400 MUSes are typically extracted. We therefore replaced the second stage of CAMUS with other hypergraph transversal computation tools, MTminer [16] and shd [22]. Both tools proved to be two orders of magnitude faster on our problems; MTminer is slightly faster but it uses significantly more memory than shd.

Our experimental evaluation is focused purely on a comparison of the clausal-resolution based prover TRP++ [17], which has previously been shown to perform well in a number of studies [17,25], and our new simplified resolution-based prover STRP. While a more comprehensive comparison featuring other proof methods similar to [25,24] would provide interesting results it is beyond the scope of this current work. Notice however that we re-use some benchmark problems from previous studies [17,25], which evaluated the performance of TRP++ against other systems, thus the performance of STRP compared with other systems can be derived from published results and our comparison. All experiments were conducted on PC with an Intel Core i5-2500K 3.30GHz CPU, with 16GB of

[1] Available at `http://www.csc.liv.ac.uk/~rmw/STRP.html`

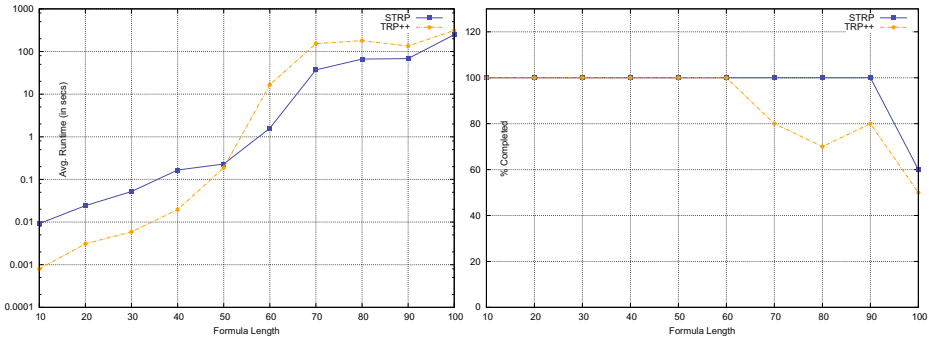


Fig. 5. Results for Rozier Benchmarks Runtimes (left) and % Tests completed (right)

RAM running Scientific Linux 6.3. As TRP++ and STRP both operate on inputs in DSNF, time taken to translate input formulae to DSNF has not been taken into account.

Random benchmarks. The first experiment involved two classes of semi-random benchmark formulae, C^1_{ran} and C^2_{ran} introduced in [18]. In previous experiments [17] TRP++ performed extremely fast (less than 0.1 on most problems), so we increased the size of the problems using the following parameters for the random formulae: $n = 48, k = 6$ and $p = 0.5$ where n is the number of propositional variables and k determines the number of distinct random variables chosen, with the polarity of each literal determined by the probability p. The results (Fig. 4) show STRP performing very consistently on both sets of problems irrespective of their size. A remarkable STRP performance on C^1_{ran} can be explained by the fact that all step clauses of C^1_{ran} problems are of the form $\mathbf{true} \Rightarrow \bigcirc(L_1 \vee ... \vee L_k)$, which are then all rewritten as a single step clause by the optimisation described above. In case of C^2_{ran}, the number of step clauses in the random formulae of different size remains fairly constant while the initial and eventuality parts increase in size and complexity. These results demonstrate the usefulness of the optimisations described above as well as suggest that the STRP performance mainly depends on the size of the step part of the input formula rather than on the size of other parts.

Another set of random benchmarks is sourced from [24]. This set of benchmarks has been first used to compare model checking approaches in [24] and as part of a more complete comparison of PLTL provers [25]. We used benchmarks with the following parameters $n = 5, p = 0.95$ and $l = 10a \ldots 100$, where n is the number of variables, p is the probability of choosing temporal operator and l is the length of the formula. The method used to generate the random formulae does not allow one to directly link the length of the formula with the number of step clauses. For example there are some problems of length $n = 90$ with well over 200 step clauses whereas some problems of length $n = 100$ contain only 100 step clauses. This variation in the number of step clauses helps to account for the variable performance, particularly for STRP, as shown in Fig. 5.

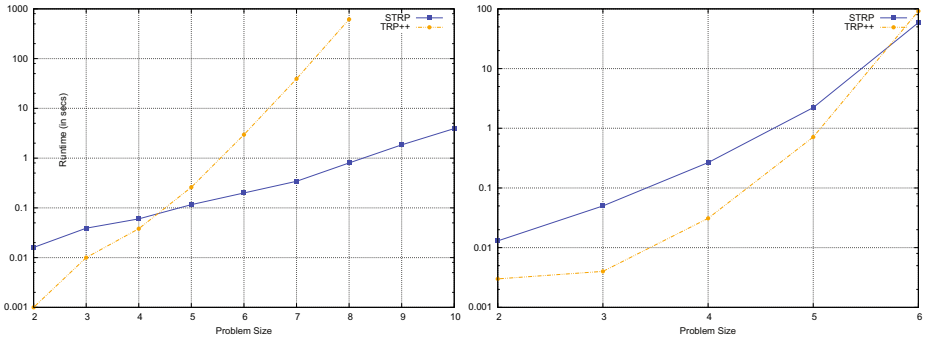


Fig. 6. O2 formulae (left) and phltl formulae (right)

The results also demonstrate that, while both system exhibit a similar dynamics with the growth of the formula size, on more complex problems ($l = 60$ and more), STRP is 5 to 10 times faster than TRP++. The lines converge due to timeouts. The plot of percentage of tests completed within the 1800s time limit (Fig. 5 right) shows that the number of tests TRP++ is able to complete within the time limit starts to drop after $l = 60$ whereas STRP only shows a decline only after $l = 90$.

Crafted benchmarks. Crafted benchmarks [25] are sets of PLTL formulae that have specifically been designed to trigger an exponential behaviour of PLTL solvers. Both TRP++ and STRP perform well on the O1 family and on the 'pattern' formulae [25] with TRP++ spending 37 seconds on the hardest Rformula1000, which contains 1998 sometime clauses and 3996 step clauses, and STRP spending 128 seconds. Problems of such large size can only be solved due the the fact that they are all trivially satisfiable or trivially unsatisfiable. For example, none of the 'pattern' formulae contain occurrences of negated propositions. Thus, simplified temporal resolution does not generate anything new from the input problem and clausal resolution generates very few (1002 in case of Rformula1000) new clauses. The extra time taken by STRP is due to I/O overhead passing information to and from the MUS extractor.

The families of O2 and phltl formulae are more challenging for both systems. STRP solves two more O2 problems within the $1,800$ second time limit; both systems show a consistent behaviour on phltl benchmarks as shown in Fig. 6. Both systems timeout on problems of size larger than given in the graphs.

TLC Cache Coherence benchmarks. Temporal Logic with Cardinality Constraints (TLC) is an extension of PLTL with global constraints on temporal interpretations, which has been introduced in [10] to capture real-world problems. An example of a TLC constraint is $\{p, q, r\}^{=1}$, which requires that exactly one proposition from the set of $\{p, q, r\}$ is true at any moment of time. The expressive power of TLC is the same as PLTL as the constraints can be captured by temporal formulae. In our example, the constraint is captured by $\Box(p \vee q \vee r)$, $\Box(\neg p \vee \neg q)$, $\Box(\neg p \vee \neg r)$, $\Box(\neg q \vee \neg r)$. It has been argued that specialised tools are

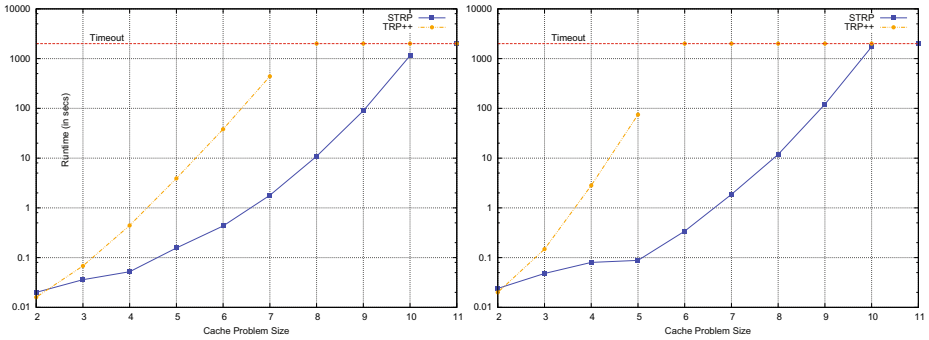


Fig. 7. Cache Coherence Results m(left) sm(right)

needed for practical reasoning in presence of cardinality constraints since PLTL formulae that capture them are too large and complex for existing provers [10].

PLTL representations of TLC formulae provide an interesting set of problems for our comparison as, due to the global nature of constraints, temporal formulae capturing such constraints contribute only the the universal part of DSNF. We use two families of TLC formulae introduced in [10] capturing verification conditions on a cache coherence protocol with n-processes, 'm': no two processes can simultaneously be in state m; and 'sm': it is not possible for one process to be in state s and another in state m. The problems feature an increasing number of processes; the number of transitions between the states is small but the set of constraints is large and complex. As a result, the PLTL representation of the original problem has a comparatively small number of step clauses but a very large number of universal clauses. As shown in Fig. 7, STRP outperforms TRP++ completing several more problems within the 1800s time limit.

Verification benchmarks. The Anzu verification benchmarks used in [3,25] provide a good counter example to the Cache Coherence problems and are particularly difficult for STRP. The smallest example from this dataset (genbuf/spec1) takes STRP 416.266s whereas TRP++ only takes 0.236s. These problems feature a moderate number of step clauses (the smallest containing 36 step clauses); however, we did find an interesting characteristic on the small number of problems we were able to run. The benchmarks produce a very large number of MCSes (43 738 on average) which are then reduced to a very small number of MUSes (58 on average) this means both stages of the MUS enumeration process take significantly longer than on other datasets explored in this work.

Performance Degradation. To evaluate how the performance of each solver changes over time, we let both systems run for 60 000s on a difficult random Rozier problem (P0.5N1L190). We captured for TRP++ the number of resolvents and universal clauses generated (both total and non-redundant) and for STRP the number of universal clauses generated (both total and non-redundant). For TRP++ this data was captured at regular intervals and for STRP we recorded a data point at each iteration of the main procedure.

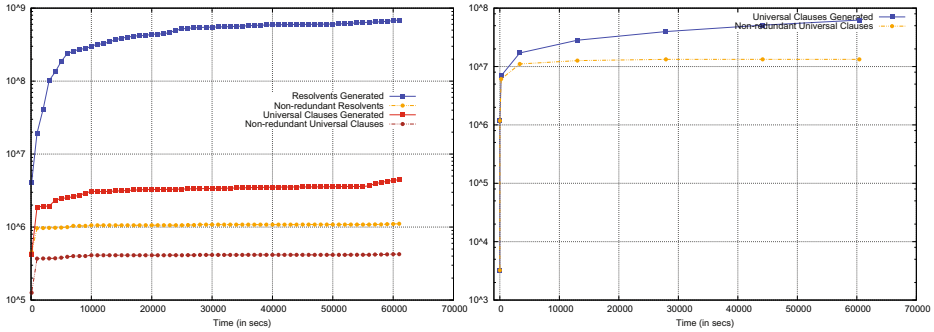


Fig. 8. TRP++ results (left) and STRP results (right)

The results (Fig. 8) show that TRP++ generates far fewer non-redundant universal clauses (125973) within the first 5 seconds of computation than STRP (1180608). Moreover, TRP++'s performance slows down noticeably at this point whereas STRP continues generating new clauses before stabilising at well over 13 million universal clauses. These numbers are not directly comparable as the systems utilise different calculi. In particular, clause-level redundancy elimination in TRP++ can be responsible for fewer non-redundant clauses being retained. However, in both calculi only universal and initial clauses contribute to the refutation of the given problem. STRP shows quite remarkable performance as it generates significantly more non-redundant universal clauses in a much shorter timeframe than TRP++.

Notice also that all formulae derived by STRP are added to the universal part, thus the search space remains constant throughout the run, which is not the case for TRP++.

5 Conclusions and Future Work

In this paper we have investigated a new approach to PLTL reasoning based on reductions to MUS enumeration. Despite the simplicity of the approach, our prototype implementation proved to perform very well on a significant number of benchmarks. Our new system performed especially well on problems having a relatively small number of step clauses but larger number of universal clauses.

Closest to our approach is bounded model checking [2], which can also establish satisfiability of PLTL formulae. However, in bounded model checking propositional formulae represent bounded-depth traces of a system and SAT solvers are used to check their realisability, while in our approach MUSes are used to facilitate resolutional proof search. Recent developments in bounded model checking include incremental inductive reasoning [4] and counting [5], which both can handle unbounded problems. The extraction of labelled superposition proofs from bounded system traces is explored in [26]. Reasoning procedures for modal logics with reductions to SAT have also been investigated in [13,14].

There are a number of possible ways to improve the performance of STRP, which constitute future work. At the moment, we use an MUS enumerating procedure as a black box. Consecutive calls of AllMus can return already known MUSes, which are then discarded as redundant. One can reuse information from the previous runs of AllMus to avoid generation of redundant MUSes. This will require modifying the MUS extractor. On larger CNF instances the MUS enumeration procedure can take a significant amount of time to return the complete set of all MUSes. It may be possible to return MUSes as and when they are derived. This facility may be useful in the SRes on unsatisfiable problems as successful application of the termination rules may be possible without the need to generate all MUSes. Finally, it would be interesting to more thoroughly investigate the impact of optimisations reducing the size of the step part on the performance of our system.

References

1. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. 25(2), 97–116 (2012)
2. Biere, A.: Bounded model checking. In: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 457–481. IOS Press (2009)
3. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Automatic hardware synthesis from specifications: A case study. In: Proceedings of DATE 2007, pp. 1–6. IEEE (2007)
4. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
5. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: Proceedings of FMCAD 2012, pp. 52–59. IEEE (2012)
6. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (1999)
7. Degtyarev, A., Fisher, M., Konev, B.: A simplified clausal resolution procedure for propositional linear-time temporal logic. In: Egly, U., Fermüller, C. (eds.) TABLEAUX 2002. LNCS (LNAI), vol. 2381, pp. 85–99. Springer, Heidelberg (2002)
8. Degtyarev, A., Fisher, M., Konev, B.: Monodic temporal resolution. ACM Trans. Comput. Log. 7(1), 108–150 (2006)
9. Dixon, C.: Using Otter for temporal resolution. In: Advances in Temporal Logic, pp. 149–166. Kluwer (2000)
10. Dixon, C., Konev, B., Fisher, M., Nietiadi, S.: Deductive temporal reasoning with constraints. Journal of Applied Logic (2012)
11. Fisher, M., Dixon, C., Peim, M.: Clausal temporal resolution. ACM Transactions on Computational Logic 2(1), 12–56 (2001)
12. Fisman, D., Kupferman, O., Sheinvald-Faragy, S., Vardi, M.Y.: A framework for inherent vacuity. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 7–22. Springer, Heidelberg (2009)
13. Giunchiglia, E., Tacchella, A., Giunchiglia, F.: SAT-based decision procedures for classical modal logics. J. Autom. Reasoning 28(2), 143–171 (2002)
14. Giunchiglia, F., Sebastiani, R.: A SAT-based decision procedure for ALC. In: Proceedings of KR 1996, pp. 304–314. Morgan Kaufmann (1996)
15. Halpern, J.Y., van der Meyden, R., Vardi, M.Y.: Complete axiomatizations for reasoning about knowledge and time. SIAM J. Comput. 33(3), 674–703 (2004)

16. Hébert, C., Bretto, A., Crémilleux, B.: A data mining formalization to improve hypergraph minimal transversal computation. Fundamenta Informaticae 80(4), 415–433 (2007)
17. Hustadt, U., Konev, B.: TRP++ 2.0: A temporal resolution prover. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 274–278. Springer, Heidelberg (2003)
18. Hustadt, U., Schmidt, R.A.: Scientific benchmarking with temporal logic decision procedures. In: Proceedings of KR 2002, pp. 533–546. Morgan Kaufmann (2002)
19. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning 40(1), 1–33 (2008)
20. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer (1992)
21. Marques-Silva, J.: Computing minimally unsatisfiable subformulas: State of the art and future directions. Multiple-Valued Logic and Soft Computing 19(1-3), 163–183 (2012)
22. Murakami, K., Uno, T.: Efficient algorithms for dualizing large-scale hypergraphs. CoRR abs/1102.3813 (2011)
23. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings POPL 1989, pp. 179–190. ACM (1989)
24. Rozier, K., Vardi, M.: LTL satisfiability checking. International Journal on Software Tools for Technology Transfer 12(2), 123–137 (2010)
25. Schuppan, V., Darmawan, L.: Evaluating LTL satisfiability solvers. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 397–413. Springer, Heidelberg (2011)
26. Suda, M., Weidenbach, C.: A PLTL-prover based on labelled superposition with partial model guidance. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 537–543. Springer, Heidelberg (2012)
27. Tansel, A.U., Clifford, J., Gadia, S.K., Jajodia, S., Segev, A., Snodgrass, R.T.: Temporal Databases: Theory, Design, and Implementation. Benjamin/Cummings (1993)
28. Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. Journal of Computer and System Sciences 32(2), 183–219 (1986)
29. Williams, R., Konev, B.: Simplified temporal resolution using SAT solvers. In: Proceedings of ARW 2012, pp. 9–10. The University of Manchester (2012)
30. Wolper, P.: The tableau method for temporal logic: An overview. Logique et Analyse 28, 119–152 (1985)

InKreSAT: Modal Reasoning via Incremental Reduction to SAT

Mark Kaminski[1] and Tobias Tebbi[2]

[1] Department of Computer Science, University of Oxford, UK
[2] Saarland University, Saarbrücken, Germany

Abstract. InKreSAT is a prover for the modal logics K, T, K4, and S4. InKreSAT reduces a given modal satisfiability problem to a Boolean satisfiability problem, which is then solved using a SAT solver. InKreSAT improves on previous work by proceeding incrementally. It interleaves translation steps with calls to the SAT solver and uses the feedback provided by the SAT solver to guide the translation. This results in better performance and allows to integrate blocking mechanisms known from modal tableau provers. Blocking, in turn, further improves performance and makes the approach applicable to the logics K4 and S4.

1 Introduction

InKreSAT is a prover for the modal logics K, T, K4, and S4 [3] that works by encoding modal formulas into SAT. The idea of a modal prover based on SAT encoding has previously been explored by Sebastiani and Vescovi [15]. While building on the same basic idea, InKreSAT extends the approach in [15] in several ways. Rather than encoding the entire modal formula in one go and then running a SAT solver on the resulting set of clauses, InKreSAT interleaves encoding phases with calls to an incremental SAT solver. If the SAT solver returns unsatisfiable, the initial modal problem is unsatisfiable, so no further encoding needs to be done. Otherwise, the SAT solver returns a propositional model of a partial encoding of the modal formula, which is used by InKreSAT to guide further encoding steps. While InKreSAT is the first system that decides modal satisfiability by incremental encoding into SAT, similar ideas have been explored for semi-decision procedures for first-order [7] and higher-order logic [4].

To deal with transitivity as it occurs in K4 and S4, and to further improve the overall performance of InKreSAT, we extend our basic approach by blocking (see, e.g., [11]).

We evaluate InKreSAT, confirming the effectiveness of our incremental approach compared to Sebastiani and Vescovi's one-phase approach. InKreSAT also proves competitive with state-of-the-art modal tableau provers.

InKreSAT is implemented in OCaml and employs the SAT solver MiniSat [5] (v2.2.0). The source code of InKreSAT and the benchmark problems used in the evaluation are available from www.ps.uni-saarland.de/~kaminski/inkresat.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 436–442, 2013.

2 Reduction to SAT

We now present the SAT encoding that underlies InKreSAT. We restrict ourselves to the case of multimodal K. An alternative, more detailed presentation of (a variant of) the encoding can be found in [15].

We distinguish between *propositional variables* (denoted p, q) and *relational variables* (denoted r). From these variables, the *formulas* of K can be obtained by the following grammar: $\varphi, \psi \ ::= \ p \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \langle r\rangle\varphi \mid [r]\varphi$. We call formulas of the form $\langle r\rangle\varphi$ *diamonds* and formulas of the form $[r]\varphi$ *boxes*.

We assume a countably infinite set of *prefixes* (denoted σ, τ) and a strict total order $\prec$ on prefixes. We call pairs $\sigma : \varphi$ *prefixed formulas*. We assume an injective function that maps every prefixed diamond $\sigma : \langle r\rangle\varphi$ to a prefix $\tau_{\sigma:\langle r\rangle\varphi}$ such that $\sigma \prec \tau_{\sigma:\langle r\rangle\varphi}$. The SAT encoding underlying InKreSAT is based on the following tableau calculus for K (working on formulas in negation normal form).

$$(\neg)\ \frac{\sigma : \varphi,\ \sigma : {\sim}\varphi}{\otimes} \qquad (\wedge_i)\ \frac{\sigma : \varphi_1 \wedge \varphi_2}{\sigma : \varphi_i}\ i \in \{1,2\} \qquad (\vee)\ \frac{\sigma : \varphi_1 \vee \varphi_2}{\sigma : \varphi_1 \mid \sigma : \varphi_2}$$

$$(\Diamond)\ \frac{\sigma : \langle r\rangle\varphi}{\tau_{\sigma:\langle r\rangle\varphi} : \varphi} \qquad (\Box)\ \frac{\sigma : [r]\varphi,\ \sigma : \langle r\rangle\psi}{\tau_{\sigma:\langle r\rangle\psi} : \varphi}$$

In the formulation of $(\neg)$, we write ${\sim}\varphi$ for the negation normal form of $\neg\varphi$, while the symbol $\otimes$ stands for the empty conclusion that closes a branch.

It can be shown that the tableau calculus is sound and complete with respect to the relational semantics of K (in fact, the calculus yields a decision procedure for K). In other words, a formula φ of K is satisfiable if and only if there is a maximal tableau rooted at $\sigma : \varphi$ (for an arbitrary prefix σ) that has an open branch (we use the terms "tableau" and "(tableau) branch" as in [6]).

Literals (denoted l) are possibly negated propositional variables. We define $\overline{\neg p} := p$ and $\overline{p} := \neg p$. We assume an injective function that maps every prefixed formula $\sigma : \varphi$ to a literal $l_{\sigma:\varphi}$ such that $l_{\sigma:\varphi} = \bar{l}_{\sigma:{\sim}\varphi}$. Note that all of the above rules have the form $\frac{\sigma:\varphi_1,\ldots,\sigma:\varphi_m}{\tau:\psi_1 \mid \cdots \mid \tau:\psi_n}$ where $m \in \{1,2\}$ and $n \in \{0,1,2\}$. Thus, we can use the following mapping to assign a clause to every instance of a rule.

$$\frac{\sigma : \varphi_1, \ldots, \sigma : \varphi_m}{\tau : \psi_1 \mid \cdots \mid \tau : \psi_n} \quad \rightsquigarrow \quad \bar{l}_{\sigma:\varphi_1} \vee \cdots \vee \bar{l}_{\sigma:\varphi_m} \vee l_{\tau:\psi_1} \vee \cdots \vee l_{\tau:\psi_n}$$

The mapping can be lifted to tableaux as demonstrated by the following example:

$\sigma : (p \vee q) \wedge \neg p$			$\rightsquigarrow$	$l_{\sigma:(p\vee q)\wedge\neg p}$
$\sigma : p \vee q$		$(\wedge_1)$	$\rightsquigarrow$	$\bar{l}_{\sigma:(p\vee q)\wedge\neg p} \vee l_{\sigma:p\vee q}$
$\sigma : \neg p$		$(\wedge_2)$	$\rightsquigarrow$	$\bar{l}_{\sigma:(p\vee q)\wedge\neg p} \vee l_{\sigma:\neg p}$
$\sigma : p$	$\sigma : q$	$(\vee)$	$\rightsquigarrow$	$\bar{l}_{\sigma:p\vee q} \vee l_{\sigma:p} \vee l_{\sigma:q}$
$\otimes$		$(\neg)$	$\rightsquigarrow$	$\bar{l}_{\sigma:p} \vee \bar{l}_{\sigma:\neg p}$ (redundant)

Note that the prefixed formula $\sigma : (p \vee q) \wedge \neg p$ at the root of the tableau is mapped to a unit clause since it is considered an assumption rather than a consequence of a tableau rule application. The last clause, which corresponds

Input: a formula φ_0
Variables: Agenda $:= \{\sigma_0 : \varphi_0\}$ (for some arbitrary but fixed prefix σ_0)
while Agenda $\neq \emptyset$ do:
1. for all $\sigma : \varphi \in$ Agenda do:
   if φ is a diamond then add $C_\sigma\varphi \cup \{ B_\sigma\psi\varphi \mid \sigma : \psi$ processed, ψ box $\}$ to SAT solver
   else if φ is a box then add $\{ B_\sigma\varphi\psi \mid \sigma : \psi$ processed, ψ diamond $\}$ to SAT solver
   else if φ is a conjunction then add $C^1_\sigma\varphi$ and $C^2_\sigma\varphi$ to SAT solver
   else add $C_\sigma\varphi$ to SAT solver
2. run SAT solver
3. if SAT solver returns unsat then return unsat
   else Agenda $:= \{ \sigma : \varphi \mid \sigma : \varphi$ pending, $l_{\sigma:\varphi}$ true in model returned by SAT solver $\}$

return sat

Fig. 1. InKreSAT: basic algorithm

to the application of $(\neg)$ to $\sigma : p$ and $\sigma : \neg p$, is redundant since our mapping of prefixed formulas to literals already ensures that $l_{\sigma:p}$ and $l_{\sigma:\neg p}$ are contradictory.

Thus, every tableau can be mapped to a set of Boolean clauses. It can be shown that the set is satisfiable if and only if the tableau has an open branch (a variant of the claim is shown in [15]). The encoding can be extended to T, K4, and S4 by suitably extending the underlying tableau calculus (see [6]).

3 Basic Algorithm

The basic algorithm underlying InKreSAT interacts with an incremental SAT solver by adding new clauses to the solver and periodically running the solver on the clauses added so far. If the solver returns satisfiable, it also returns a satisfying model that is used in selecting new clauses to be added.

The *premise of a clause* C, where C corresponds to an instance of a tableau rule $\frac{\sigma:\varphi_1,\dots,\sigma:\varphi_m}{\tau:\psi_1|\cdots|\tau:\psi_n}$, consists of the literals $l_{\sigma:\varphi_1}, \dots, l_{\sigma:\varphi_m}$. We call a prefixed formula $\sigma : \varphi$ *processed* if $l_{\sigma:\varphi}$ occurs in the premise of a clause added to the SAT solver. Otherwise, we call $\sigma : \varphi$ *pending*, but only if (1) $l_{\sigma:\varphi}$ occurs in a clause added to the SAT solver and (2) φ is not of the form p or $\neg p$. We exclude formulas $\sigma : p$ and $\sigma : \neg p$ because they require no further processing: since our mapping of prefixed formulas to literals takes care of trivial inconsistencies, we never generate clauses for the rule $(\neg)$.

Let φ be a disjunction or a diamond. We write $C_\sigma\varphi$ for the clause corresponding to the instance of $(\vee)$ or $(\Diamond)$, resp., that has $\sigma : \varphi$ as its unique premise (e.g., $C_\sigma(\langle r\rangle p) = \bar{l}_{\sigma:\langle r\rangle p} \vee l_{\tau_{\sigma:\langle r\rangle p}:p}$). If φ is a conjunction, we write $C^i_\sigma\varphi$ $(i \in \{1, 2\})$ for the clause corresponding to the instance of $(\wedge_i)$ that has $\sigma : \varphi$ as its premise. Finally, we write $B_\sigma\varphi\psi$, where φ is a box and ψ a diamond, for the clause corresponding to the instance of $(\Box)$ that has $\sigma : \varphi$ and $\sigma : \psi$ as its premises.

The basic algorithm (restricted to K) is shown in Fig. 1. It maintains an agenda consisting of pending prefixed formulas that are true in the model returned by a preceding invocation of the SAT solver (initially, the agenda contains the input formula). Every formula $\sigma : \varphi$ on the agenda is processed by adding clauses with $l_{\sigma:\varphi}$ in their premise (after which $\sigma : \varphi$ becomes processed).

4 Blocking

Blocking is a technique commonly used with tableau calculi to achieve termination in the presence of transitive relations or background theories [11]. Even when it is not required for termination, blocking can improve the performance of tableau-based decision procedures [10]. Blocking typically restricts the applicability of the rule ($\Diamond$). An application of ($\Diamond$) to a prefixed diamond on a branch is blocked if one can determine that the successor prefix that would be introduced by the application is subsumed by some prefix that is already on the branch.

Because of the close correspondence between the translational method underlying InKreSAT and modal tableau calculi, blocking is necessary to make our translation terminating in the presence of transitive relations.

Extending the one-phase approach in [15] with blocking is problematic since the approach has no explicit representation of tableau branches. Known blocking techniques are all designed to work on a single tableau branch at a time. Blocking across branches typically destroys the correctness of a tableau system.

In our case, however, the propositional model used to guide the translation in step 3 of the main loop in Fig. 1 yields a suitable approximation of a tableau branch—the formulas whose corresponding literals are true in the model. We can show that blocking restricted to these formulas preserves the correctness of our procedure. To explore the impact of blocking, we extend the basic algorithm by a variant of anywhere blocking [1] (with ideas from pattern-based blocking [12]).

Unlike with tableau provers [10], blocking in InKreSAT can cause considerable overhead. After every run of the SAT solver, the data structures needed for blocking may have to be recomputed from scratch because models returned by two successive runs of the solver may differ in an unpredictable way. To avoid the recomputation, we must be able to guarantee that the model returned by the solver is an extension of the previously computed model. This leads us to a final refinement of our procedure, called *model extension* (MX). We make the SAT solver always first search for extensions of the existing model by adding all literals true in the model to the input of the solver (as unit clauses). If the solver finds an extension of the model, we proceed without recomputing the data structures for blocking. Otherwise, we run the solver once again, now without the additional clauses, and recompute the data structures from scratch. The goal of MX is to reduce the overhead of blocking, thus increasing its effectiveness. On the other hand, MX can cause more calls to the solver, which may decrease performance.

5 Evaluation and Conclusions

We evaluate the effects of incremental translation to SAT and blocking by running InKreSAT in four different modes: a "one phase" mode, where, like in [15], the encoding is generated in one go, a "no blocking" mode, where clause generation is performed incrementally, but blocking is switched off, a "no MX" mode, where blocking is enabled, but MX is disabled, and the default mode, where both blocking and MX are enabled. Besides, we include the results from four other

Table 1. Results on the LWB benchmarks for K (left) and S4 (right)

Subclass	InKreSAT (default)	InKreSAT (no MX)	InKreSAT (no blocking)	InKreSAT (one phase)	Spartacus	FaCT++	K2SAT	*SAT
branch_n	12	12	13	4	9	10	**15**	12
branch_p	**18**	15	14	4	10	9	16	**18**
d4_n	**21**	**21**	7	6	**21**	**21**	6	**21**
d4_p	**21**	**21**	13	8	**21**	**21**	9	**21**
dum_n	**21**	**21**	**21**	17	**21**	**21**	19	**21**
dum_p	**21**	**21**	**21**	16	**21**	**21**	18	**21**
lin_n	**21**	**21**	**21**	**21**	**21**	**21**	**21**	13
path_n	14	**21**	6	6	**21**	**21**	13	**21**
path_p	12	**21**	8	7	**21**	**21**	14	**21**
ph_n	**21**	**21**	**21**	**21**	**21**	12	**21**	11
ph_p	**9**	**9**	**9**	**9**	8	7	**9**	8
t4p_n	**21**	**21**	7	4	**21**	**21**	4	**21**
t4p_p	**21**	**21**	13	8	**21**	**21**	8	**21**

Subclass	InKreSAT (default)	InKreSAT (no MX)	Spartacus	FaCT++
branch_n	**11**	**11**	9	6
md_n	8	9	**21**	10
md_p	3	4	**9**	4
ipc_n	9	11	**21**	10
ipc_p	8	11	**21**	9
path_n	4	9	16	**21**
path_p	5	10	17	**21**
ph_n	**13**	11	10	8
ph_p	**9**	**9**	5	6
s5_n	14	**20**	16	19

provers. (1) K2SAT, Sebastiani and Vescovi's [15] implementation of their one-phase translational approach. We used K2SAT in conjunction with MiniSat 2.0, which is directly integrated into the system (the integrated solution outperformed a setup using MiniSat 2.2.0, which is used by InKreSAT). We used the options `-j -u -v -w` recommended by the authors. (2) *SAT [16] (v1.3), a reasoner for the description logic $\mathcal{ALC}$. *SAT also integrates SAT technology, but does so in a way that is different from our approach. It uses a SAT solver only for propositional reasoning, while modal reasoning is handled by a conventional tableau calculus. (3) FaCT++ [17] (v1.6.1), an established reasoner for the web ontology language OWL 2 DL. (4) Spartacus [10] (v1.1.3), an efficient prover for the hybrid logic $\mathcal{H}(E, @)$. K2SAT and *SAT are included because they implement related approaches while FaCT++ and Spartacus are supposed to indicate the state of the art in automated reasoning for modal logic. Except for K2SAT, all provers are compiled and run with the default settings (unlike in [8]).

We perform the tests on a Pentium 4 2.8 GHz, 1 GB RAM, with a 60s time limit per formula (the same setup as in [10]). **Table 1:** The K and S4 problem sets from the Logic Work Bench (LWB) benchmarks [2]. LWB is widely used for measuring the performance of modal reasoners (e.g., in [8,10,15]). LWB is the only suite available to us that includes S4 problems. For each subclass that was not solved in its hardest instance (21) by every system, Table 1 displays the hardest instance that could be solved (the best results set in bold). The evaluation on the S4 problems is limited to systems and configurations of InKreSAT that can cope with transitivity. **Fig. 2**, upper half**:** Randomly generated 3CNF_K [8] formulas of modal depth 2, 4, and 6 (45 problems each, 135 in total; see [10] for details). The selection allows us to see how performance depends on modal depth. We plot the number of instances that could be solved against time. The plot on the left-hand side compares the four different modes of InKreSAT, while on the right we compare InKreSAT to the other provers. **Fig. 2**, lower half**:** A subset of the TANCS-2000 [14] Unbounded Modal QBF (MQBF) benchmarks for K complemented by randomly generated modalized MQBF formulas [13] (800 problems

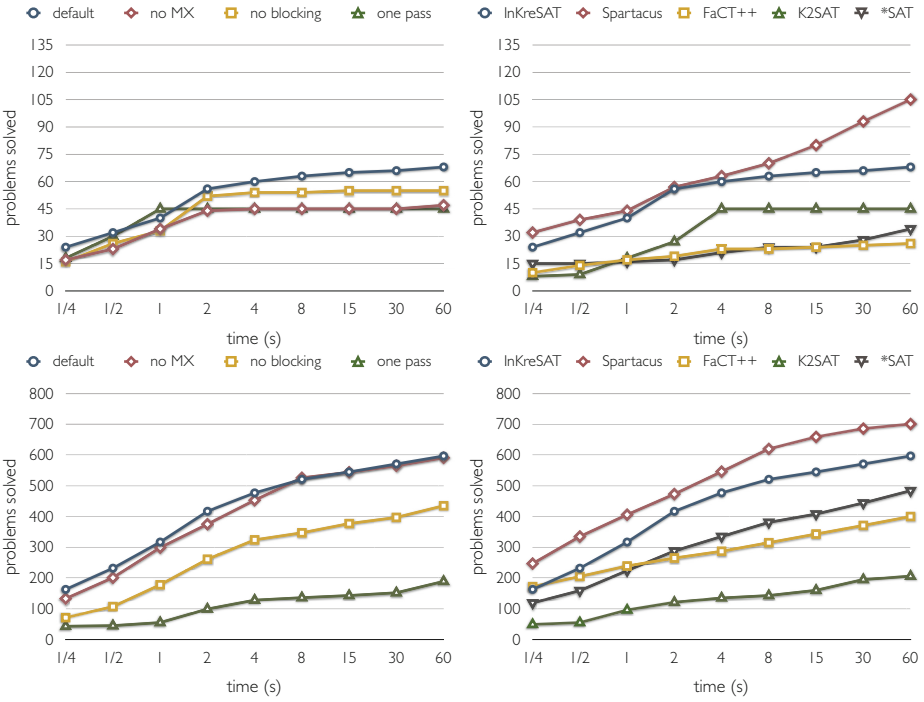


Fig. 2. Results on 3CNF$_K$ (upper half) and MQBF formulas (lower half)

in total). In selecting the MQBF problems, we follow [9], but restrict ourselves to the "easy/medium" and "medium" problem classes because of our time limit of 60s. For the same reason, we leave out the harder subclasses of non-modalized "medium" problems (keeping only the problems with V=4, see [9,13,14,10]).

We observe that incremental translation and blocking both lead to considerable performance gains on all benchmarks. With MX, the results are mixed. On LWB, InKreSAT generally performs better without MX. On MQBF, MX makes little difference. On 3CNF$_K$, however, it is MX that makes blocking efficient and allows InKreSAT to solve more formulas of high modal depth (solving 45/13/10 formulas of depth 2/4/6, resp., compared to 45/9/1 without blocking). Without MX, the overhead caused by blocking actually diminishes performance (to 45/2/0). Compared to the other systems, InKreSAT proves competitive, solving a number of problems that cannot be solved by others, and displaying the arguably best results (without MX) on LWB-K. Note also that in the "one phase" mode, the behavior of InKreSAT expectedly resembles that of K2SAT, K2SAT being slightly faster because of additional optimizations that do not work with incremental translation. A notable weakness of InKreSAT as compared to tableau provers is a faster degradation of performance with increasing modal depth (on LWB-S4 and especially on 3CNF$_K$, where, e.g., Spartacus solves 40/38/27 problems of depth 2/4/6). We attribute the faster degradation to the higher overhead of blocking in the present setting and to a lack of a more

efficient heuristic to guide clause generation. Solving these problems, as well as extending the approach to more expressive logics (e.g., logics with nominals or converse modalities), are interesting directions for future work.

Acknowledgments. This work was partially supported by the EPSRC project Score!.

References

1. Baader, F., Buchheit, M., Hollunder, B.: Cardinality restrictions on concepts. Artif. Intell. 88(1-2), 195–213 (1996)
2. Balsiger, P., Heuerding, A., Schwendimann, S.: A benchmark method for the propositional modal logics K, KT, S4. J. Autom. Reasoning 24(3), 297–317 (2000)
3. Blackburn, P., van Benthem, J., Wolter, F. (eds.): Handbook of Modal Logic. Elsevier (2007)
4. Brown, C.E.: Encoding higher-order theorem proving to a sequence of SAT problems. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 147–161. Springer, Heidelberg (2011)
5. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
6. Fitting, M.: Modal proof theory. In: Blackburn et al. [3], pp. 85–138
7. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. In: LICS 2003, pp. 55–64. IEEE Computer Society Press (2003)
8. Giunchiglia, E., Giunchiglia, F., Tacchella, A.: SAT-based decision procedures for classical modal logics. J. Autom. Reasoning 28(2), 143–171 (2002)
9. Giunchiglia, E., Tacchella, A.: A subset-matching size-bounded cache for testing satisfiability in modal logics. Ann. Math. Artif. Intell. 33(1), 39–67 (2001)
10. Götzmann, D., Kaminski, M., Smolka, G.: Spartacus: A tableau prover for hybrid logic. In: Bolander, T., Braüner, T. (eds.) M4M-6. ENTCS, vol. 262, pp. 127–139. Elsevier (2010)
11. Horrocks, I., Hustadt, U., Sattler, U., Schmidt, R.: Computational modal logic. In: Blackburn, et al. [3], pp. 181–245
12. Kaminski, M., Smolka, G.: Terminating tableau systems for hybrid logic with difference and converse. J. Log. Lang. Inf. 18(4), 437–464 (2009)
13. Massacci, F.: Design and results of the Tableaux-99 non-classical (modal) systems comparison. In: Murray, N.V. (ed.) TABLEAUX 1999. LNCS (LNAI), vol. 1617, pp. 14–18. Springer, Heidelberg (1999)
14. Massacci, F., Donini, F.M.: Design and results of TANCS-2000 non-classical (modal) systems comparison. In: Dyckhoff, R. (ed.) TABLEAUX 2000. LNCS, vol. 1847, pp. 52–56. Springer, Heidelberg (2000)
15. Sebastiani, R., Vescovi, M.: Automated reasoning in modal and description logics via SAT encoding: The case study of $K_m/\mathcal{ALC}$-satisfiability. J. Artif. Intell. Res. 35, 343–389 (2009)
16. Tacchella, A.: *SAT system description. In: Lambrix, P., Borgida, A., Lenzerini, M., Möller, R., Patel-Schneider, P. (eds.) DL 1999, vol. 22. CEUR-WS.org (1999)
17. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006)

BV2EPR: A Tool for Polynomially Translating Quantifier-Free Bit-Vector Formulas into EPR*

Gergely Kovásznai, Andreas Fröhlich, and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract. Bit-precise reasoning is essential in many applications of Satisfiability Modulo Theories (SMT). In recent years, efficient approaches for solving fixed-size bit-vector formulas have been developed. Most of these approaches rely on bit-blasting. In [1], we argued that bit-blasting is not polynomial in general, and then showed that solving quantifier-free bit-vector formulas (QF_BV) is NEXPTIME-complete. In this paper, we present a tool based on a new polynomial translation from QF_BV into Effectively Propositional Logic (EPR). This allows us to solve QF_BV problems using EPR solvers and avoids the exponential growth that comes with bit-blasting. Additionally, our tool allows us to easily generate new challenging benchmarks for EPR solvers.

1 Introduction

Bit-precise reasoning over bit-vector logics is important for many practical applications of Satisfiability Modulo Theories (SMT), particularly for hardware and software verification. Examples of state-of-the-art SMT solvers with support for fixed-sized bit-vector logics are Boolector, MathSAT, STP, Z3, and Yices. All these solvers rely on *bit-blasting* in order to translate bit-vector formulas into propositional logic (SAT). The result is then checked by a SAT solver.

In practice, e.g. in the SMT-LIB [2], the BTOR [3], and the Z3 format, the bit-widths in bit-vector formulas are encoded as binary, decimal, or hexadecimal numbers, i.e., a *logarithmic encoding* is used. In [1], we proved that the encoding of bit-widths affects the complexity of the decision problem of bit-vector logics. In particular, logarithmic encoding makes the quantifier-free fragment QF_BV2 NEXPTIME-complete.[1] Thus, bit-blasting is *not polynomial* in general. For a polynomial reduction, the target logic has to be NEXPTIME-hard.

In this paper, we introduce our new tool BV2EPR. BV2EPR translates QF_BV formulas into Effectively Propositional Logic (EPR), which is NEXPTIME-complete [4], by using a new (polynomial) reduction. This is in contrast to existing translations in [5,6], which produce exponential EPR formulas in general, as we will point out in Sect. 2.1. We give some experimental results in Sect. 4 with the EPR solver iProver.

* Supported by FWF, NFN Grant S11408-N23 (RiSE).

[1] In [1], we introduced the notation QF_BV1 resp. QF_BV2 for QF_BV using a *unary* resp. a *logarithmic*, actually without loss of generality, *binary encoding*.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 443–449, 2013.

2 Preliminaries

We assume the usual syntax for QF_BV. A bit-vector term t of bit-width n ($n \in \mathbb{N}$, $n \geqslant 1$) is denoted by $t^{[n]}$. An atomic term can be either (a) a bit-vector *constant* $c^{[n]}$, where $c \in \mathbb{N}$, $0 \leqslant c < 2^n$; or (b) a bit-vector *variable* $v^{[n]}$. Compound terms and formulas can contain the usual bit-vector *operators* (c.f. SMT-LIB [2]), like e.g. bitwise operators, shifts, arithmetic operators, relational operators, etc. The decision problem for QF_BV is NExpTime-complete [1].

EPR, known as the Bernays-Schönfinkel class, is a NExpTime-complete fragment of first-order logic [4]. It corresponds to the set of first-order formulas that, written in prenex form, contain (a) no function symbol of arity greater than 0; and (b) no existential quantifier within the scope of a universal quantifier. After Skolemization, existential variables turn into constants (i.e., function symbols of arity 0), and quantifiers can be omitted. Consequently, an EPR *atom* can be defined as an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n and each t_i is either a (universal) variable or a constant.

2.1 Existing Translations

In [5], encodings of hardware verification problems with bit-vectors into first-order logic are proposed. In particular, an encoding into EPR is given and called the *relational encoding* [6], since bit-vectors are modeled as unary predicates. These predicates are over bit-indices, represented by dedicated constants. For instance, the ith bit of a bit-vector $x^{[n]}$, $0 \leqslant i < n$, is represented by the atom $p_x(bitInd_i)$, where $bitInd_i$ is a constant. Note that for QF_BV2, such a translation might introduce exponentially many constants, since bit-widths like n are encoded logarithmically. The so-called *range-aware* relational encoding in [6], furthermore, introduces exponentially many assertions into the EPR formula in general, e.g., atoms $less_k(bitInd_i)$ for all $0 \leqslant i < k$. Finally, not all the QF_BV bit-vector operators are addressed by the relational encoding, but only *equality*[2]. All the *arithmetic operators* are assumed to be synthesized/bit-blasted in the verification front-end [6], potentially leading to an exponential blowup already before the actual encoding. In [5], an abstraction of *shifts* is proposed, which is, however, basically the same as bit-blasting. Consequently, the relational encoding is exponential in general, in constrast with our translation in Sect. 3.1.

3 The Tool

BV2EPR takes a QF_BV formula in SMT2 format as input, and outputs an EPR clause set in TPTP format. The tool is implemented in C and available at [7]. The architecture of BV2EPR can be seen in Fig. 1, consisting of the following modules:

Parser. The Parser is Boolector's SMT2 parser.

[2] Bitwise operators could be handled in a similar way.

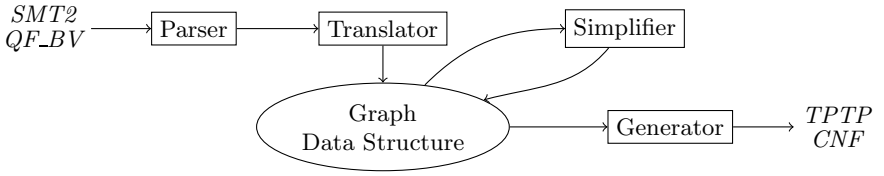


Fig. 1. The architecture of BV2EPR

Translator. The Translator provides an interface accessed by the Parser, in order to deal with the SMT2 QF_BV operators. This module builds a graph data structure, in which each bit-vector operation is modeled by an EPR predicate. Predicates are represented by shared nodes in the graph data structure. A node for a predicate p stores, besides other data, the functional definition of p as an EPR clause set. With each of these clauses, an argument list $i_{n-1}, \ldots, i_0$ for p is stored, indicating that this clause is part of the functional definition of the EPR atom $p(i_{n-1}, \ldots, i_0)$. Such a clause is realized as a list of EPR literals, each of which contains a reference to a predicate p' and an argument list for p'.

Simplifier. The graph constructed by the Translator is a good basis for various simplifications. Note that only polynomial simplification steps are acceptable. Among others, we implemented two kinds of simplification, both proposed in [8]: (a) *unused definition elimination* and (b) *non-growing definition inlining*.

Generator. Out of the (simplified) graph, this module generates a TPTP clause set. Since the graph might contain cycles, the Generator detects and avoids them. Due to the construction of the graph data structure, clauses can be extracted directly, i.e., no additional approach for clause generation is needed.

3.1 The Translator

We briefly sketch the (polynomial) reduction of QF_BV to EPR used by the Translator, without striving for completeness. As it will turn out, the target logic of this reduction is actually not general EPR, but rather its fragment which uses only two constants, 0 and 1. We call this fragment EPR2.[3] To each bit-vector term of bit-width n, a dedicated $\lceil \log_2 n \rceil$-ary EPR2 predicate is introduced and assigned. For example, a term $x^{[32]}$ is represented by a 5-ary predicate p_x. Since p_x is an EPR2 predicate, each of its arguments can be either 0, 1, or a universal variable. For instance, the atom $p_x(1, 1, 0, 0, 1)$ represents the 25th bit of x, since $25_{10} = 11001_2$. Using universal variables as arguments makes it possible to represent several bits by a single EPR2 formula; for instance, the atom $p_x(i_4, i_3, i_2, i_1, 0)$ represents all even bits of x.

Bitwise Operators. Translating bitwise operators is quite natural. We demonstrate the translation for *bitwise or* (denoted by $|$): Given a term $x^{[2^n]} \mid y^{[2^n]}$, where x and y are bit-vector terms, to which the predicates p_x and p_y have already been assigned, respectively. We need to specify each bit of

[3] The Herbrand universe of EPR2 can be considered as the Boolean domain.

the resulting bit-vector as the disjunction of the corresponding bits of x and y. We introduce a new predicate p_{or}, and give the following functional definition:

$$p_{or}(i_{n-1},\ldots,i_0) \;\Leftrightarrow\; p_x(i_{n-1},\ldots,i_0) \vee p_y(i_{n-1},\ldots,i_0)$$

Addition. Given a term $x^{[2^n]} + y^{[2^n]}$, let us first rewrite it to the following bit-vector equations, where $\oplus$ denotes *bitwise xor*, & *bitwise and*, and $\ll$ *left shift*.

$$add^{[2^n]} = x^{[2^n]} \oplus y^{[2^n]} \oplus cin^{[2^n]} \quad (1)$$
$$cin^{[2^n]} = cout^{[2^n]} \ll 1 \quad (2)$$
$$cout^{[2^n]} = (x^{[2^n]} \;\&\; y^{[2^n]}) \mid (x^{[2^n]} \;\&\; cin^{[2^n]}) \mid (y^{[2^n]} \;\&\; cin^{[2^n]}) \quad (3)$$

Note that Eqn. (1) and (3) only contain bitwise operators (and equality). Therefore, both can be translated into EPR2 as introduced previously. Only Eqn. (2), which contains *shift by 1*, has to be handled differently.

We introduce a helper predicate *succ* which will represent the fact that a bit-index j is the successor of a bit-index i, i.e., $j = i + 1$. Since i is represented by an EPR2 argument list $i_{n-1},\ldots,i_0$ and, similarly, j by $j_{n-1},\ldots,j_0$, the $2n$-ary predicate $succ(i_{n-1},\ldots,i_0,\ j_{n-1},\ldots,j_0)$ can be defined by n facts:

$$\begin{gathered} succ(i_{n-1},\ldots,i_3,i_2,i_1,0,\ \ i_{n-1},\ldots,i_3,i_2,i_1,\ 1) \\ succ(i_{n-1},\ldots,i_3,i_2,\ 0,1,\ \ i_{n-1},\ldots,i_3,i_2,\ 1,\ 0) \\ succ(i_{n-1},\ldots,i_3,\ 0,\ 1,1,\ \ i_{n-1},\ldots,i_3,\ 1,\ 0,\ 0) \\ \vdots \\ succ(0,1,\ldots,1,\ 1,0,\ldots,0) \end{gathered}$$

Using this helper predicate, Eqn. (2) can be translated into EPR2 as follows:

$$\begin{gathered} \neg p_{cin}(0,\ldots,0) \\ succ(i_{n-1},\ldots,i_0,\ j_{n-1},\ldots,j_0) \;\Rightarrow\; (p_{cin}(j_{n-1},\ldots,j_0) \;\Leftrightarrow\; p_{cout}(i_{n-1},\ldots,i_0)) \end{gathered}$$

This kind of adder can be adapted to represent other arithmetic operators like *unary minus* and *subtraction*. In BV2EPR, all the relational operators, like *equality* and *unsigned less than*, are also represented by such an adapted adder.

Shifts. Shifts are translated into EPR2 by applying *barrel shift*. For instance, given a term $x^{[2^n]} \ll y^{[2^n]}$, for all bit-indices i, $0 \leqslant i < n$, the ith bit of y is checked: if it is 1, then *left shift by* 2^i has to be done.

$$\begin{gathered} \neg p_y(0,\ldots,0) \;\Rightarrow\; \left(p^0_{shl}(i_{n-1},\ldots,i_0) \Leftrightarrow p_x(i_{n-1},\ldots,i_0)\right) \\ \left(\begin{array}{c} p_y(0,\ldots,0) \;\wedge \\ succ(i_{n-1},\ldots,i_0,\ j_{n-1},\ldots,j_0) \end{array}\right) \;\Rightarrow\; \left(p^0_{shl}(j_{n-1},\ldots,j_0) \Leftrightarrow p_x(i_{n-1},\ldots,i_0)\right) \\ \neg p_y(0,\ldots,0,1) \;\Rightarrow\; \left(p^1_{shl}(i_{n-1},\ldots,i_0) \Leftrightarrow p^0_{shl}(i_{n-1},\ldots,i_0)\right) \\ \left(\begin{array}{c} p_y(0,\ldots,0,1) \;\wedge \\ succ(0,i_{n-1},\ldots,i_1,\ 0,j_{n-1},\ldots,j_1) \end{array}\right) \;\Rightarrow\; \left(p^1_{shl}(j_{n-1},\ldots,j_1,i_0) \Leftrightarrow p^0_{shl}(i_{n-1},\ldots,i_0)\right) \\ \vdots \end{gathered}$$

Multiplication. The Translator applies a *shift-and-add* approach for translating a term $x^{[2^n]} \cdot y^{[2^n]}$. We generate 2^n subproducts of bit-width 2^n, and represent all of them by a single $2n$-ary predicate p_{mul}: the ith bit of the jth subproduct is represented by the atom $p_{mul}(j_{n-1}, \ldots, j_0,\ i_{n-1}, \ldots, i_0)$.

First, the $(2^n - 1)$th subproduct is computed, by checking the most significant bit of y: if it is 0, this subproduct is set to 0; otherwise, it is set equal to x.

$$\begin{aligned} \neg p_y(1, \ldots, 1) \ &\Rightarrow\ \neg p_{mul}(1, \ldots, 1,\ i_{n-1}, \ldots, i_0) \\ p_y(1, \ldots, 1) \ &\Rightarrow\ (p_{mul}(1, \ldots, 1,\ i_{n-1}, \ldots, i_0) \Leftrightarrow p_x(i_{n-1}, \ldots, i_0)) \end{aligned}$$

The jth subproduct, $0 \leqslant j < 2^n - 1$, is computed by checking the jth bit of y: if it is 0, then the $(j+1)$th subproduct has to be *shifted left by 1* (represented by the predicate p_{shl}); otherwise, the shifted subproduct and x have to be *added* (represented by p_{add}).

$$\begin{pmatrix} \neg p_y(j_{n-1}, \ldots, j_0) \ \wedge \\ succ(j_{n-1}, \ldots, j_0,\ j'_{n-1}, \ldots, j'_0) \end{pmatrix} \Rightarrow \begin{pmatrix} p_{mul}(j_{n-1}, \ldots, j_0,\ i_{n-1}, \ldots, i_0) \ \Leftrightarrow \\ p_{shl}(j'_{n-1}, \ldots, j'_0,\ i_{n-1}, \ldots, i_0) \end{pmatrix}$$

$$\begin{pmatrix} p_y(j_{n-1}, \ldots, j_0) \ \wedge \\ succ(j_{n-1}, \ldots, j_0,\ j'_{n-1}, \ldots, j'_0) \end{pmatrix} \Rightarrow \begin{pmatrix} p_{mul}(j_{n-1}, \ldots, j_0,\ i_{n-1}, \ldots, i_0) \ \Leftrightarrow \\ p_{add}(j'_{n-1}, \ldots, j'_0,\ i_{n-1}, \ldots, i_0) \end{pmatrix}$$

The final product is given by $p_{mul}(0, \ldots, 0,\ i_{n-1}, \ldots, i_0)$.

Polynomiality and Correctness. All above translation steps are polynomial in the input size since they are polynomial in the number of atoms and logarithmic in their bit-width. Formally showing correctness exceeds the scope of this paper and is part of future work. We also investigated correctness empirically by exhaustively testing consistency of the solving results by Boolector and BV2EPR+iProver, for each bit-vector operation, up to a certain bit-width.

4 Benchmarks and Experiments

Solving QF_BV formulas in general is NEXPTIME-complete [1]. However, certain families of QF_BV formulas are in NP, under certain restrictions on the bit-widths. We called this kind of families *bit-width bounded* [1]. Since solving EPR formulas is NEXPTIME-complete, our translation fits well to families which are *not* bit-width bounded. In [1], two examples of this kind were given: (a) `QF_BV/brummayerbiere3/mulhs`*bw* represents instances of computing the *high-order half of product* problem, parameterized by the bit-width of multiplicands (*bw*); (b) `QF_BV/bruttomesso/lfsr/lfsr`*t_bw_n* formalizes the behaviour of a *linear feedback shift register* [9]. We further propose two new benchmark families that are *not bit-width bounded*: (a) `add2n`*bw* describes how bit-vectors of bit-width 2*bw* can be added by using two adders for bit-vectors of bit-width *bw*. (b) `addmul`*bw* checks, whether the sum of two bit-vectors of bit-width *bw* can differ from their product.

In order to demonstrate the exponential blow-up of bit-blasting, in contrast to our translation into EPR, we used the bit-blaster Synthebtor, part of the

Table 1. Evaluation for the original SMT2 file

bmark	bw	smt2	btor	Boolector	aig	cnf	Lingeling	epr	iProver
mulhs	$8^{\ddagger}$	947	1K	10.3s	3K	44K	9.0s	45K	1m 44s
	$16^{\ddagger}$	959	1K	TO	12K	205K	TO	55K	TO
	$64^{\ddagger}$	982	2K	TO	221K	4M	TO	78K	TO
lfsr_2_bw_16	$63^{\ddagger}$	6K	9K	0.2s	64K	258K	0.7s	56K	18.0s
	$127^{\ddagger}$	7K	9K	1.2s	139K	545K	1.3s	61K	1m 14s
	1023	7K	11K	5.1s	1M	5M	4.7s	74K	TO
	8191	7K	18K	2m 37s	11M	43M	3m 10s	89K	TO
add2n	2^5	452	455	0.0s	3K	25K	0.1s	12K	1m 21s
	2^6	456	671	0.1s	7K	53K	0.7s	13K	TO
	2^{12}	484	8K	3m 5s	549K	4M	1m 28s	21K	TO
addmul	2^7	149	99	0.2s	174K	3M	2.4s	8K	0.1s
	2^9	149	99	2.7s	3M	58M	3m 22s	11K	0.1s
	2^{11}	151	103	TO	48M	1G	TO	13K	0.1s

Table 2. Evaluation for the simplified SMT2 file

bmark	bw	smt2	btor	Boolector	aig	cnf	Lingeling	epr	iProver
mulhs	$8^{\ddagger}$	2K	804	9.8s	3K	42K	8.1s	63K	1m 48s
	$16^{\ddagger}$	2K	956	TO	11K	197K	TO	77K	TO
	$64^{\ddagger}$	2K	1K	TO	215K	4M	TO	110K	TO
lfsr_2_bw_16	$63^{\ddagger}$	126K	59K	0.5s	81K	254K	0.9s	156K	3.0s
	$127^{\ddagger}$	126K	59K	0.6s	174K	540K	1.4s	158K	9.5s
	1023	126K	60K	7.0s	1M	5M	5.1s	165K	9m 21s
	8191	126K	67K	46.1s	13M	43M	TO	173K	TO
add2n	2^5	1K	575	0.0s	4K	25K	0.1s	17K	23.6s
	2^6	1K	671	0.1s	9K	53K	0.7s	18K	5m 0s
	2^{12}	2K	9K	2m 42s	711K	4M	1m 16s	32K	TO
addmul	2^7	239	75	0.2s	174K	3M	2.5s	8K	0.1s
	2^9	239	75	2.8s	3M	58M	1m 40s	11K	0.1s
	2^{11}	241	79	TO	48M	1G	TO	13K	0.1s

Boolector distribution, to generate AIGER files and DIMACS (CNF) files out of BTOR files. Tab. 1 summarizes these results, when *word-level rewriting* in Boolector is switched off. We give the file sizes (in bytes) in all formats and additionally provide the runtimes of Boolector (for SMT2), Lingeling (for CNF), and iProver (for EPR), using a timeout of 10 minutes.

In order to test the effect of word-level rewriting, we added a module to Boolector which reads an SMT2 file, performs rewriting, and outputs the simplified SMT2 file. In Tab. 2, we give the results for the simplified SMT2 files.

‡ Official SMT-LIB benchmarks.

5 Conclusion

We presented BV2EPR, a tool for polynomially translating QF_BV into EPR. The motivation for our tool lies in previous work [1], where we have shown QF_BV to be NEXPTIME-complete. Thus, bit-blasting QF_BV to SAT, as it is usually done in current SMT solvers, results in exponentially larger formulas in general. Previous translations from QF_BV into EPR also apply bit-blasting on certain operators and introduce exponentially many constants resp. constraints in the general case [5,6]. In contrast to this, the Translator used in BV2EPR always produces EPR formulas of polynomial size.

After discussing BV2EPR, we evaluated the size of the formulas produced by our tool and compared it to other commonly used formats. Our results show that the overhead in size is rather small when translating QF_BV into EPR, while all other formats often suffer from exponential blow-up as soon as the bit-widths in the input formula grow larger. However, our results also show that the runtime of iProver on the generated EPR formulas is usually worse compared to the runtime of Boolector on the original QF_BV formula or the one of Lingeling after bit-blasting has been applied. Nevertheless, the evaluation also shows that there exist benchmarks where iProver is faster. While it is probably still possible to improve EPR solvers on this kind of instances, formulas generated by BV2EPR can also help providing challenging benchmarks for current state-of-the-art solvers. The tool BV2EPR is available at [7].

References

1. Kovásznai, G., Fröhlich, A., Biere, A.: On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In: Proc. SMT 2012, pp. 44–55 (2012)
2. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: Version 2.0. In: Proc. SMT 2010, Edinburgh, UK (2010)
3. Brummayer, R., Biere, A., Lonsing, F.: BTOR: bit-precise modelling of word-level problems for model checking. In: BPR 2008, pp. 33–38. ACM, New York (2008)
4. Lewis, H.R.: Complexity results for classes of quantificational formulas. J. Comput. Syst. Sci. 21(3), 317–353 (1980)
5. Khasidashvili, Z., Kinanah, M., Voronkov, A.: Verifying equivalence of memories using a first order logic theorem prover. In: FMCAD 2009, pp. 128–135 (2009)
6. Emmer, M., Khasidashvili, Z., Korovin, K., Voronkov, A.: Encoding industrial hardware verification problems into effectively propositional logic. In: FMCAD 2010, pp. 137–144 (2010)
7. BV2EPR project page, `http://fmv.jku.at/bv2epr/`
8. Hoder, K., Khasidashvili, Z., Korovin, K., Voronkov, A.: Preprocessing techniques for first-order clausification. In: FMCAD 2012, pp. 44–51 (2012)
9. Bruttomesso, R., Sharygina, N.: A scalable decision procedure for fixed-width bit-vectors. In: Proc. ICCAD 2009, pp. 13–20. IEEE (2009)

The 481 Ways to Split a Clause and Deal with Propositional Variables

Kryštof Hoder and Andrei Voronkov

University of Manchester, Manchester, UK

Abstract. It is often the case that first-order problems contain propositional variables and that proof-search generates many clauses that can be split into components with disjoint sets of variables. This is especially true for problems coming from some applications, where many ground literals occur in the problems and even more are generated.

The problem of dealing with such clauses has so far been addressed using either splitting with backtracking (as in Spass [14]) or splitting without backtracking (as in Vampire [7]). However, the only extensive experiments described in the literature [6] show that on the average using splitting solves fewer problems, yet there are some problems that can be solved only using splitting.

We tried to identify essential issues contributing to efficiency in dealing with splitting in resolution theorem provers and enhanced the theorem prover Vampire with new options, algorithms and datastructures dealing with splitting. This paper describes these options, algorithms and datastructures and analyses their performance in extensive experiments carried out over the TPTP library [12]. Another contribution of this paper is a calculus RePro separating propositional reasoning from first-order reasoning.

1 Introduction

In first-order theorem proving, theorem provers based on variants of resolution and superposition calculi (in the sequel simply called *resolution theorem provers*) are predominant. This is confirmed by the results of the last CASC competitions.[1] The top three theorem provers Vampire [7], E-MaLeS and E [9] are resolution-based, while the fourth one iProver [4] implements both an instance-based calculus and resolution with superposition.

Resolution theorem provers use *saturation algorithms*. They deal with a search space consisting of clauses. Inferences performed by saturation algoritms are of three different kinds:

1. *Generating inferences* derive a new clause from clauses in the search space. This new clause can then be immediately simplified and/or deleted by other kinds of inference.
2. *Simplifying inferences* replace a clause by another clause that is simpler in some strict sense.
3. *Deletion inferences* delete clauses from the search space.

[1] `http://www.cs.miami.edu/~tptp/CASC/J6/`

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 450–464, 2013.

On hard problems the search space is often growing rapidly, and simplifications and deletions consume considerable time. Performance of resolution theorem provers degrades especially fast when they generate many clauses having more than one literal (*multi-literal clauses* for short) and heavy clauses (clauses of large sizes). There are several reasons for this degradation of performance:

1. The complexity of algorithms implementing inference rules depends on the size of clauses. The extreme case are algorithms for subsumption and subsumption resolutions. These problems are known to be NP-complete and algorithms implementing them are exponential in the number of literals in clauses.
2. Storing heavy clauses requires more memory. Moreover, every literal in a clause (and sometimes every term occurring in such a literal) are normally added to one or more indexes. Index maintenance requires considerable space and time and operations on these indexes slow down significantly when the indexes become large.
3. Generating inferences applied to heavy clauses usually generate heavy clauses. Generating inferences applied to clauses with many literals usually generate clauses with many literals. For example, resolution applied to two clauses containing n_1 and n_2 literals normally gives a clause with $n_1 + n_2 - 2$ literals.

To deal with multi-literal and heavy clauses, one can simply start discarding them after some time, thus losing completeness [8]. Alternatively, one can use *splitting*. There are two kinds of splitting described in the literature: splitting with backtracking (as in Spass [14]) or splitting without backtracking (as in Vampire [7]).

2 Propositional Variables in Resolution Provers

Both kinds of splitting are implemented using introduction of propositional variables denoting components of split clauses. When many such variables are introduced, they give rise to clauses with many propositional literals. Such clauses clog up search space and slow down expensive operations, such as subsumption. Therefore, the problem of dealing with propositional literals is closely related to splitting. Apart from variables arising from splitting, propositional variables are common in many applications, for example, program analysis. They may also be introduced during preprocessing when naming is used to generate small clausal normal forms.

The resolution and superposition calculus is very efficient for proving theorems in first-order logic. In propositional logic, it is not competitive to DPLL. Suppose that we have a problem that uses both propositional and non-propositional atoms. Then treating propositional atoms in the same way as non-propositional ones results in performance problems. For example, if we use the code trees technique for implementing subsumption [13] and make no special treatment for propositional variables, it will work in the worst case in exponential time even for a pair of propositional clauses, while the best algorithms for propositional subsumption are linear.

The Calculus RePro To address the problem of dealing with propositional variables, in this section we will introduce a *calculus RePro* for dealing with clauses having propositional literals, and will illustrate some options of Vampire using this calculus. The calculus separates propositional reasoning from non-propositional.

Let us call a *pro-clause* any expression of the form $C \mid P$, where C is a clause containing no propositional variables and P is a propositional formula. Logically, this pro-clause is equivalent to $C \vee P$, so the bar sign $\mid$ can be seen as simply separating the propositional and non-propositional parts of the pro-clause. We will consider a clause containing no propositional variables as a special kind of pro-clause, in which P is a false formula.

Note that a pro-clause $C \vee P$ is not necessarily a clause, since P can be an arbitrary formula. Also, any propositional formula P can be considered a special case of a pro-clause $\Box \mid P$, where $\Box$ denotes, as usual, the empty clause. We will call any pro-clause $\Box \mid P$ *propositional*.

The calculus RePro is parametrised by an *underlying resolution calculus*. That is, for every resolution calculus on clauses we will define an instance of the calculus RePro based on this resolution calculus. However, since we are not varying the underlying calculus in this paper, we will simply speak of RePro as a calculus.

Generating Inferences. For every generating inference

$$\frac{C_1 \quad \cdots \quad C_n}{C}$$

of the resolution calculus the following is an inference rule of RePro:

$$\frac{C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n}{C \mid (P_1 \vee \ldots \vee P_n)} .$$

Simplifying Inferences. Let

$$\frac{C_1 \quad \cdots \quad C_n \quad \not{D}}{C} \tag{1}$$

be a simplifying inference of the resolution calculus. Speaking the theory of resolution, this means that C is implied by $C_1, \ldots, C_n, D$ and D is redundant with respect to $C_1, \ldots, C_n, C$. If $P_1 \vee \ldots \vee P_n \rightarrow P$ is a tautology, then the following is a simplifying inference rule of RePro:

$$\frac{C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n \quad \not{D \mid P}}{C \mid (P_1 \vee \ldots \vee P_n)} . \tag{2}$$

Deletion Inferences. Let

$$C_1 \quad \cdots \quad C_n \quad \not{D}$$

be a deletion inference of the resolution calculus, that is, D is redundant with respect to $C_1, \ldots, C_n$. If $P_1 \vee \ldots \vee P_n \rightarrow P$ is a tautology, then the following is a deletion inference of RePro:

$$C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n \quad \not{D \mid P}.$$

Completeness. It is not hard to derive soundness and completeness of RePro assuming the same properties of the underlying resolution calculus, however completeness here means something different from completeness in the theory of resolution. The reason for this difference is that RePro contains essentially no rules for dealing with the

propositional part of clauses. In the completeness theorem below, we assume knowledge of the theory of resolution [1,5]. Also, we do not specify the underlying calculus, for example, the calculus used in Vampire can be used.

Theorem 1 (Completeness). *Let $S_0, S_1, S_2, \ldots$ be a fair sequence of sets of pro-clauses such that S_0 is unsatisfiable. Then there exists $i \geq 0$ such that the set of propositional pro-clauses in S_i is unsatisfiable too.*

The proof is omitted here. Note that this theorem implies that the proof-search in RePro can be carried out by using any standard fair saturation algorithm to perform RePro inferences corresponding to the rules of the underlying calculus plus unsatisfiability checking for the propositional part. This is how it is implemented in Vampire.

To implement such an algorithm for RePro on top of a standard implementation of the resolution calculus one needs to address the following questions:

(q1) representation of the propositional part of pro-clauses;
(q2) representation of propositional pro-clauses (which can be different from the representation of the propositional part of pro-clauses);
(q3) unsatisfiability checking for sets of propositional pro-clauses;
(q4) efficient simplification rules for pro-clauses.

There are some other implementation details to be addressed. For example, the inference selection process in saturation algorithms usually depends on the weights of clauses (which is usually their size measured in the number of symbols). One can use different size measures for pro-clauses, especially when their propositional parts are not necessarily clauses. This adds one more question:

(q5) pro-clause selection.

Before discussing possible answers we will introduce some other rules that can be used in RePro.
Propositional tautology deletion is a deletion rule of RePro formulated as follows:

$$\frac{D \mid P}{}\,,$$

where P is a tautology.
The merge rule of RePro is formulated as follows:

$$\frac{C \mid P_1 \quad C \mid P_2}{C \mid (P_1 \wedge P_2)}$$

Note that so far this is the only rule that introduces propositional formulas other than clauses.
The merge subsumption rule of RePro is formulated as follows:

$$\frac{C \mid P_1 \quad D \mid P_2}{D \mid (P_1 \wedge P_2)}\,,$$

where C subsumes D. This rule can also introduce propositional formulas that are not clauses.

The Calculus ReProR One can also define simplifying rules on pro-clauses in an alternative way. Namely, the modification is as follows. Consider a simplifying rule (1) of the underlying resolution calculus. Then the following can be considered as a simplifying inference rule:

$$\frac{C_1 \mid P_1 \quad \cdots \quad C_n \mid P_n \quad \cancel{D \mid P}}{C \mid (P_1 \vee \ldots \vee P_n) \quad D \mid (P_1 \vee \ldots \vee P_n \to P)}\ .$$

One can see that the previously defined simplifying rule (2) is a special case of this one, since, if $P_1 \vee \ldots \vee P_n \to P$ is a tautology, the second inferred clause can be removed. One can also reformulate the deletion rules in the same way. We will denote the resulting calculus ReProR (the refined RePro). Note that the simplification rules of the refined calculus introduce non-clauses in the propositional part.

The advantage of the alternative formulation of simplification and deletion rules is that one clause can be simplified away into a tautology using a sequence of simplifying or deletion rules impossible in the standard formulation of RePro. For example, a clause $A \vee B \mid (p \wedge q)$ is redundant in the presence of $A \mid p$ and $B \mid q$ using the following sequence of subsumption deletion rules:

$$\dfrac{B \mid q \quad \dfrac{A \mid p \quad \cancel{A \vee B \mid (p \wedge q)}}{\cancel{A \vee B \mid (p \to (p \wedge q))}}}{A \vee B \mid (q \to (p \to (p \wedge q)))}$$

whose conclusion is a tautology.

3 Splitting

In very simple terms, splitting is based on the following idea. Suppose that S is a set of clauses and $C_1 \vee C_2$ a clause such that the variables of C_1 and C_2 are disjoint. Then the set $S \cup \{C_1 \vee C_2\}$ is unsatisfiable if and only if both $S \cup \{C_1\}$ and $S \cup \{C_2\}$ are unsatisfiable. There is more than one way to implement splitting. Before discussing them let us introduce some definitions.

Recall that a clause is a disjunction $L_1 \vee \ldots \vee L_n$ of *literals*, where a literal is an atomic formula or a negation of an atomic formula. A literal or clause is *ground* if it contains no occurrences of variables. In the context of splitting we consider a clause as a set of its literals. In other words, we assume that clauses do not contain multiple occurrences of the same literal and clauses equal up to permutation of literals are considered equal. Let $C_1, \ldots, C_n$ be clauses such that $n \geq 2$ and all the C_i's have pairwise disjoint sets of variables. Then we say that the clause $C \stackrel{\text{def}}{=} C_1 \vee \ldots \vee C_n$ is *splittable* into *components* $C_1, \ldots, C_n$. We will also say that the set $C_1, \ldots, C_n$ is a *splitting* of C. For example, every ground multi-literal clause is splittable. There may be more than one way to split a clause, however there is always a unique splitting such that each component C_i is non-splittable: we call this splitting *maximal*. It is easy to see that a maximal splitting has the largest number of components and every splitting with the largest number of components is the maximal one. There is a simple algorithm for finding the maximal splitting of a clause [6], which is, essentially, the union-find algorithm.

Splittable clauses appear especially often when theorem provers are used for software verification and static analysis. Problems used in these applications usually have a large number of ground clauses (coming from program analysis) and a small number of non-ground clauses (for example, an axiomatisation of memory or objects).

There are essentially two ways of using splitting in a first-order resolution theorem prover. One is splitting with backtracking as implemented in Spass [14] and another *splitting without backtracking* [6]. Each of them is described in the next subsections, where we also point out potential efficiency problems associated with each kind of splitting.

When we discuss the use of splitting in resolution theorem provers, it is very important to understand how the use of splitting affects other components of such provers. The efficiency and power of modern resolution theorem provers comes from two techniques: *redundancy elimination* (see [1] for the theory and [8] for the implementation aspects) and *term indexing* [10]. Another component important for understanding efficiency is the saturation algorithm and especially the clause selection algorithm used to implement this algorithm.

Redundancy Elimination. Unlike backtracking algorithms used in DPLL, saturation algorithms are backtracking-free. When clauses are simplified or deleted, these simplifications and deletions do not have to be undone. On the contrary, some forms of splitting may require backtracking.

Term Indexing. Even when simplifications are used, the search space can quickly grow to hundreds of thousands of clauses. To perform inferences on such a large search space efficiently, theorem provers maintain several indexes storing information about terms and clauses. These indexes make it easier to find candidates for inferences. In some cases inferences can be performed only by using the relevant index, without retrieving clauses used for these inferences. The number of different indexes in theorem provers varies and can be as many as about 10. Frequent insertion and deletion in an index can affect performance of a theorem prover. A typical example is when a theorem prover generates an equality $a = b$ between two constants and uses it to rewrite a into b. For nearly all indexing techniques used in the resolution theorem provers, every term and clause containing a must be removed from all indexes and a new term containing b inserted in them again. Doing this single simplification step on an indexes set with 100,000 clauses can take a very long time.

Clause Selection. Selection of generating inferences in resolution theorem provers is implemented using *clause selection*. For selection, clauses are put in one or more priority queues and selected based on their priorities. Normally, the majority of selected clauses are taken from the available clauses of the smallest weight.

The use of splitting may heavily affect all these parts of the saturation algorithm implementation: redundancy elimination, term indexing and clause selection. Let us discuss this in more detail in the rest of this section.

Splitting without Backtracking. Splitting without backtracking [6] can be implemented using a naming technique. Suppose we have a splittable clause $C_1 \vee \ldots \vee C_n$ with components $C_1, \ldots, C_n$. We introduce new propositional variables $p_1, \ldots, p_{n-1}$ to "name" the first $n - 1$ components. That is, we introduce them together with definitions $p_i \leftrightarrow C_i$. Then we use the rule

$$\frac{\cancel{C_1 \vee \ldots \vee C_n}}{C_1 \vee \neg p_1 \qquad \ldots \qquad C_{n-1} \vee \neg p_{n-1} \qquad C_n \vee p_1 \vee \ldots p_{n-1}}$$

If the same components appear more than once in a splittable clause, their names can be reused. In fact, they should be reused, as shown in [6].

The advantage of this approach is that we do not need to perform any backtracking, which spares us the costs of inserting and deleting clauses from indexes. The only additional cost to the implementation of saturation is an index of components required to reuse names. Such an index is used for all kinds of splitting in Vampire. Checking whether one component is a variant of another is equivalent to the graph isomorphism problem, see [6], yet it practice maintaining and using this index requires the time negligible compared to the overall running time.

Splitting without backtracking is very efficient on some problems but may also be very inefficient, since it can introduce thousands of propositional variables and long clauses containing these variables.

Another drawback of this method is that simplifying inferences are not being performed "across branches." For example, when we split clause $f(a) = a \vee q(a)$, we obtain $f(a) = a \vee p$, so we cannot use the equality $f(a) = a$ to simplify expressions such as $q(f(a))$ by demodulation. In the case of splitting with backtracking, we would obtain the unit clause $f(a) = a$, and all the demodulation simplifications would be performed (though at the cost of having to backtrack them later).

Splitting with Backtracking. Splitting with backtracking is based on the idea of the DPLL splitting. It uses the labelled clause calculus introduced in [3]. We will first describe the use of labels and then show how it can be captured by a variation of the RePro calculus.

When we have a splittable clause $C_1 \vee C$, where C_1 is a minimal component, it is first replaced by C_1, and when we derive a contradiction that follows from C_1, we (well, almost) backtrack to the point of the split and introduce the rest of the clause C. If C_1 is ground (and therefore a literal), we may also add, in the spirit of DPLL, $\neg C_1$ at this point. (Whether we do so is controlled by a Vampire option.)

To implement this technique, we assign a label to every split that we perform, and augment each clause C by a set of split labels on which it depends. Each newly derived clause depends on a set of labels that is the union of the sets belonging to its parents. When a clause is deleted, we need to examine the labels of the clauses which justified the deletion. If the deletion was justified by some labels on which the deleted clause itself does not depend, we do not delete the clause, but rather keep it aside to be restored if we backtrack beyond the label that justified its deletion.

Our implementation of the backtracking splitting can be captured by the RePro calculus, if we restrict the inferences that can be performed at any given point, and introduce a different treatment for simplification inferences.

We do not use any of the RePro rules that would introduce non-clauses into the propositional part. The propositional parts of pro-clauses are therefore clauses, and their literals correspond to the labels that the clause has assigned in the labelled calculus.

We are maintaining a partial model M which is initially empty and to which we add propositional literals corresponding to active splits. At each point we restrict inferences of the RePro calculus to the pro-clauses whose propositional part is not satisfied by the model M.

The split labels are seen as fresh propositional variables and the label introduction at splitting $C_1 \vee C$ can be viewed as naming C_1 with a propositional variable. More precisely, when we split $C_1 \vee C \mid P$ and use the name p_1 for C_1, we add pro-clauses $C_1 \mid (\neg p_1 \vee P)$ and $C \mid (p_1 \vee P)$. We also make a note that p_1 *depends* on every propositional variable in P and add p_1 into the partial model M.

At this point, having p_1 in M keeps the clause $C \mid (p_1 \vee P)$ from participating in any inferences for now. Also, the original clause $C_1 \vee C \mid P$ is subsumed by the newly introduced $C_1 \mid (\neg p_1 \vee P)$ modulo the partial model M.

Clause simplification and restoring upon backtracking is the part that does not fit well into the RePro calculus and for which we need to introduce a special treatment. Among the pro-clauses with propositional parts not satisfied by the partial model M we perform simplifications as we would have done it in the base calculus. Except that there may come a point when we will restore the simplified clause back into the search space: When a clause $C \mid P$ is simplified with $C_1 \mid P_1, \ldots, C_n \mid P_n$ as premises, the restoration of the simplified clause in the labelled calculus corresponds to the point when the formula $F \equiv (P_1 \vee \ldots \vee P_n) \rightarrow P$ becomes no longer satisfied by the partial model M. As a matter of fact, F is actually the propositional formula in one of the conclusions of simplifying inferences in the ReProR calculus. One would be therefore tempted to use the ReProR calculus to capture the splitting with backtracking. However, the problem is that the formula F is not a clause, and the labelled calculus we use to implement backtracking splitting does not easily capture general formulas using the clause labels.

We therefore rather check with each change of the model M whether the condition for restoring any of the simplified clauses does occur, and if so, we put the clause back into the saturation algorithm.

When we derive a propositional pro-clause $\Box \mid Q$, we select an atom p in the clause $Q \equiv Q' \vee \neg p$ so that no atom in Q' *depends* on it (if there is more than one such atom, we choose arbitrarily). Let us remind that there is only one pro-clause with a positive occurrence of p — the clause $C \mid (p \vee P)$ which we introduced after splitting $C_1 \vee C \mid P$. This clause became inactive as we added p into the partial model M, so it could not spread the literal p any further. Now we resolve the pro-clauses $C \mid (p \vee P)$ and $\Box \mid Q' \vee \neg p$ on the atom p to obtain $C \mid (Q' \vee P)$, delete the clause $C \mid (p \vee P)$ and replace p in M with $\neg p$.

Note that we have removed p from M which means that the originally split clause $C_1 \vee C \mid P$ is restored, even though just to become subsumed again by $C \mid (Q' \vee P)$. Now there are two possibilities. If Q' is an empty clause, $C_1 \vee C \mid P$ will never be restored as it is subsumed by $C \mid P$ which has the same propositional part. This corresponds to the case when we have refuted the first branch of the split without any additional assumptions. If Q' is a non-empty clause, the original split clause may be restored if some of the assumptions on which we refuted the split is backtracked.

It should be noted that while we can change the polarity of a propositional variable in the model M from positive to negative, we never change it from negative to positive. Therefore, once we assign false to a propositional variable, all pro-clauses that contain it in a negative literal may be deleted.

Drawbacks. The disadvantage of this kind of splitting is that, upon backtracking, we sometimes have to delete and restore many clauses. This leads to costly index maintenance operations, and also a lot of work can be wasted.

For example, suppose we split a clause $a = b \vee C$ where the symbol b is the smallest in the simplification ordering and does not appear anywhere else in the problem, while a has many occurrences. Splitting this clause will introduce a unit clause $a = b$ and the backward demodulation will replace every occurrence of a by b, resulting in massive updates in all indexes. Since b does not appear anywhere else, the equality will not be helpful in any way, but all the rewritten clauses will depend on this split. Once we reach a refutation using the rewritten clauses, we will have to restore all the clases containing a, once again resulting in massive updates in all indexes. Also note that we may end up doing a lot of repeated work as the proof search on the branch using $a \neq b$ will be likely similar to the one on the $a = b$ branch.

4 Implementation and Parameters

In this section we describe various parameters implemented in Vampire and related to splitting and/or use of propositional variables. We also discuss these parameters in the context of the questions (q1)–(q5). These parameters and their values are summarised in Table 1.

Splitting. The main parameter controlling splitting is `splitting`. It has three values: `backtracking`, `nobacktracking` and `off`. All other options have two values: `on` and `off`.

Clauses may be split either eagerly, before they enter the passive clause container, or the splitting can be postponed until a clause is selected for activation. This is controlled by the option `split_at_activation`.

The set of split clauses can be restricted in several ways. Option `split_goal_only` restricts splitting only to goal clauses and clauses that are derived from them. Enabling `split_input_only` excludes derived clauses from splitting, allowing splits only on the clauses which were initially passed to the saturation algorithm.

A different kind of restriction is to add a requirement that both split components contain fewer positive literals than the original clause. Such splitting will lead to clauses that are closer to Horn form which allows at most one positive literal per clause. This setting is enabled by the `split_positive` option.

Splitting with Backtracking. The implementation is based on [14]. We extended it by use of time stamps and reference counters on clauses. This allows us to implement the structure for restoring clauses upon backtracking more efficiently — upon backtracking we only traverse the list of clauses that is to be restored, and let the time-stamping ensure that we never restore the same clause twice.

If `split_add_ground_negation` is enabled, upon backtracking caused by splitting a ground literal L, we add its negation $\neg L$ as a new clause. The option

Table 1. Option names, short names and values

option	short	values
general		
splitting	spl	backtracking, nobacktracking, off
split_add_ground_negation	sagn	on, off
what and when to split		
split_goal_only	sgo	on, off
split_input_only	sio	on, off
split_positive	spo	on, off
split_at_activation	sac	on, off
propositional pro-clauses		
propositional_to_bdd	ptb	on, off
sat_solver_for_empty_clause	ssec	on, off
sat_solver_with_naming	sswn	on, off
simplifications		
sat_solver_with_subsumption_resolution	sswsr	on, off
empty_clause_subsumption	ecs	on, off
bdd_marking_subsumption	bms	on, off
clause and literal selection		
nonliterals_in_clause_weight	nicw	on, off
splitting_with_blocking	swb	on, off

`nonliterals_in_clause_weight` means that the weight of each clause will be increased by the number of splits on which it depends.

Propositional Parts of Pro-Clauses. There are several possible implementations of pro-clauses with a clausal propositional part. However, variants of RePro using non-clausal propositional parts quickly lead to very complex formulas for which the only suitable data structure we could think of was ordered binary decision diagrams [2], or simply BDDs. Thus, we extended the clause objects in Vampire by a reference to the BDD representing the propositional part of a pro-clause.

Since the refined calculus ReProR requires the use of arbitrary formulas, we also used BDDs to implement this calculus. We hoped that it will be very efficient for some problems since many more clauses would be simplified away. In reality ReProR turned out to be almost dysfunctional. The refined simplification rules created ever more complex propositional formulas with very large BDDs. In many cases these BDDs quickly used all the available memory. It was also common that nearly all runtime of Vampire was consumed by BDD operations. Therefore, we decided not to use ReProR and report no results on it in this paper.

The option `propositional_to_bdd` (`q1`) chooses whether BDDs are used to store propositional parts of pro-clauses. If BDDs are not in use, we treat propositional literals in the same way as all other literals. It is a separate issue how to deal with purely propositional clauses. One can also treat them as ordinary clauses. However, one can choose to pass them to a SAT solver instead. Since every propositional clause

can be considered as a pro-clause $\Box \mid P$ with the empty non-propositional part, options for dealing with such clauses use `empty_clause` in their names. In the option `sat_solver_for_empty_clause` (q2,q3) is on, such clauses are passed to a SAT solver.

When we use BDDs for pro-clauses but not for propositional clauses, whenever we obtain a BDD for a propositional clause, we must convert this BDD to a set of clauses. The number of propositional clauses obtained from a BDD can be exponential. To cope with this problem, we added an option `sat_solver_with_naming` (q2) that would make conversion of BDDs to clauses almost linear time by introducing new propositional variables. An alternative to `sat_solver_with_naming` is the option `sat_solver_with_subsumption_resolution` which uses subsumption resolution to shorten the long clauses generated when converting BDDs to CNF without the introduction of new propositional variables.

If we decide to represent the propositional pro-clauses as BDDs, the transition from a first-order pro-clause into propositional is straightforward. We keep at most one propositional pro-clause by eagerly applying a propositional merging rule

$$\frac{\cancel{\Box \mid P} \quad \cancel{\Box \mid P'}}{\Box \mid (P \wedge P')}$$

whenever obtain a new propositional pro-clause $\Box \mid P'$. This way we know that if the set of propositional clauses becomes unsatisfiable, we will derive a propositional clause with associated $\bot$ BDD node.

For first-order pro-clauses we may decide to reflect the complexity of the propositional part in the clause selection process. To this end, enabling the option `nonliterals_in_clause_weight` in presence of BDDs increases the size of clauses by the depths of the BDD of their propositional parts.[2]

When we derive a propositional pro-clause $\Box \mid P$, clauses $C \mid P'$ such that $P \to P'$ become redundant. This follows from the RePro version of the subsumption rule as an empty clause subsumes any other clause:

$$\Box \mid P \quad \cancel{C \mid P'} \quad \text{if } P \to P'$$

It would not be feasible to make an implication check between P and the propositional part of every pro-clause present in the saturation algorithm. We have implemented two incomplete checks for subsumption by propositional pro-clauses.

The first one focuses on the premises of the derived propositional pro-clause, as there is a good chance that some of the ancestors will also have P as its propositional part. If we succeed with some of the premises, we carry on the check with their premises and further on in the derivation graph, as long as we are succeeding. This check is controlled by the option `empty_clause_subsumption`.

[2] The dag size of BDDs would probably better reflect the complexity of propositional formulas, but computing this measure is not a "local" operation on BDDs — one would need to traverse the whole BDD subgraph to count the distinct nodes. The depth of a BDD can, however, be computed by using just the depths of immediate successors. The tree size of a BDD can be computed locally as well, however it can grow exponentially with the size of the BDD.

The second of the checks uses the shared structure of the BDDs. When we derive a propositional pro-clause $\Box \mid P$, we set a *subsumed* flag in the BDD node corresponding to P. Whenever we see a first-order pro-clause to have a BDD node with the *subsumed* flag, we know it is redundant and can be deleted. Moreover, our BDD implementation is aware of this flag and attempts to "spread" the mark while performing other BDD operations. For example, when performing a disjunction operation, if one of the operands has the flag set, it will be set also for the node representing the disjunction of the operands. This subsumption algorithm is controlled by the option `bdd_marking_subsumption`.

5 Evaluation

There are all together 481 different combinations of values for the Vampire parameters related to splitting and propositional variables, so analysing the results was far from trivial. For simplicity, we will call them *splitting parameters,* though this name is a bit misleading since some of them are actually related to dealing with propositional variables.

For benchmarking we used unsatisfiable TPTP problems having non-unit clauses and rating greater than 0.2 and less than 1. Essentially, the rating is the percentage of existing provers that cannot solve a problem. For example, rating greater than 0.2 means that less than 80% of existing theorem provers can solve the problem. Likewise, rating 1 means that the problem cannot solved by the existing provers. However, the rating evaluation uses a single mode of every prover, so it is possible that the same prover can solve a problem of rating 1 using a different mode. For this reason, we also added problems of rating 1 and solvable by Vampire.[3] We excluded very large problems since for them it was preprocessing, but not other options, that affect results the most. This resulted in 4,869 TPTP problems.

To conduct the experiments, we took a Vampire strategy that is believed to be nearly the best in the overall number of solved problems, and generated the 481 variations of this strategy obtained by setting the splitting parameters to all possible values. For each of these variations, we ran it on the selected problems with a 30 seconds time limit. This resulted in 2,341,989 runs, which roughly correspond to 1.5 years of run time on a single core.

We evaluated the experiments in two different ways. First, we looked at the best overall strategies for the backtracking and non-backtracking splitting, and how many problems they solve. However, the number of solved problems for a single (even the best) setting of parameters is not the main criterion of importance for splitting parameters.

The reason for this is that it is known that problems are normally best solved by attempting them with a cocktail of strategies. The CASC [11] version of Vampire uses a sequence of strategies to solve a problem, and using such a sequence is also a recommended mode for the users. Therefore in the second part of evaluation we looked at the numbers of problems solvable only by particular settings of the splitting parameters.

[3] Solvable by Vampire means solvable with at least one of the 481 different strategies.

Table 2. Problems solved by each setting of the splitting strategy

splitting	strategies	worst	average	best
off	25	2708	2720	2737
backtracking	64	1825	2710	3143
non-backtracking	416	1756	2608	2929

Table 3. Best and worst strategies

	worst	best
`splitting`	`nobacktracking`	`backtracking`
`propositional_to_bdd`	`on`	
`split_at_activation`	`off`	`on`
`split_goal_only`	`off`	`off`
`split_input_only`	`off`	`off`
`split_positive`	`off`	`off`
`nonliterals_in_clause_weight`	`off`	`off`
`bdd_marking_subsumption`	`off`	
`empty_clause_subsumption`	`on`	
`sat_solver_for_empty_clause`	`off`	
`split_add_ground_negation`		`on`

The Best and the Worst Strategies. Only 3,598 (about 74% of all problems) were solved by at least one splitting strategy. The top-level results are summarised in Table 2. The best and the worst strategies are shown in Table 3. Some of the option values in the table are left out because not all combinations of parameters make sense. For example, for backtracking splitting we use labeled clauses, not BDDs, so all BDD related options are left out. As one can see, without splitting all strategies behave very similar, which is expected, since problems normally contain few propositional symbols. However, the use of splitting makes a very substantial difference, especially for the best strategies. For example, the best strategy using splitting solved 3143 problems versus 2737 problems solved without splitting. Another interesting point is a huge gap between the performance of the worst and the best strategies using the same kind of splitting. However, the biggest surprise for us was the fact that the best strategies used splitting with backtracking.

Importance of Particular Parameters. To determine the importance of various splitting options, we put the numbers of problems that can be solved only with a particular value of an option into Table 4. Under (a) we show the number of problems that can be solved either only by backtracking or non-backtracking splitting. The number of problems solvable only without any splitting at all is zero. This perhaps surprising result is due to the fact that splitting can be restricted using the options `split_input_only`, `split_goal_only` and `split_positive` to the extent that almost no splits are actually performed.

The cases (b)–(m) show the numbers of problems requiring a particular value of an option for some of the following cases: `off`, `backtracking`, `nobacktracking` or

Table 4. Problems solved only by a single value of an option

a) splitting

off	0
noback	128
back	198

b) split_at_activation

	on	off
back	147	73
noback	91	93
all	145	113

c) split_goal_only

	on	off
back	31	155
noback	21	207
all	17	159

d) split_input_only

	on	off
back	43	414
noback	67	302
all	33	384

e) split_positive

	on	off
back	37	262
noback	28	146
all	35	181

f) propositional_to_bdd

	on	off
off	62	45
noback	227	107
all	226	106

g) nonliterals_in_clause_weight

	on	off
off	17	11
back	55	45
noback	23	62
all	33	91

h) splitting_with_blocking

	on	off
noback	20	290

i) sat_solver_for_empty_clause

	on	off
off	8	5
noback	34	21
all	34	21

j) sat_solver_with_naming

	on	off
off	2	0
noback	22	0
all	22	0

k) sat_solver_with_subsumption_resolution

	on	off
off	2	1
noback	1	2
all	2	2

l) bdd_marking_subsumption

	on	off
off	62	45
noback	227	107
all	226	106

m) empty_clause_subsumption

	on	off
off	5	7
noback	18	46
all	18	46

n) split_add_ground_negation

	on	off
back	191	6

all. In the first three cases, the numbers for columns off and on stand for the number of problems which could be solved for the specified value of splitting only with the option enabled or disabled. More precisely, e.g. for the column off of option A we give the number of problems for which there existed values of other options so that problem was solved with option A disabled, but for all the combinations of parameters the problem was not solved when the option A was enabled. The row all gives numbers of problems where particular option value was required across all relevant splitting modes.

From the Table 4 (j) it can be seen that the use of naming in clausification of BDDs is always a good thing to do, as none of the problems required to have this option disabled. From case (n) it can be seen that it is very rarely the case that adding ground negations after refuting a splitting branch will harm, as only 6 problems are lost by enabling this setting, however 191 problems required to have this setting enabled. On the other hand, for many other options, having the possibility to enable or disable them is important, as either setting can solve problems which cannot be solved by the other.

6 Conclusion

We implemented two variants of clause splitting and many ways of implementing them in a first-order theorem prover, and through extensive experiments we have shown that the backtracking splitting in our setup gives the best performance. More importantly, we have also shown the importance of keeping a large portfolio of strategies, because a large group of problems can be solved only by a variety of different approaches, not by having only one strategy, even though performing well on average.

Aside of the extensive experimental evaluation, we also presented new families of calculi RePro and ReProR which separate propositional from first-order reasoning.

All the described parameters are supported by the current version of the Vampire theorem prover which is available for download at `http://www.vprover.org`.

References

1. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 2, vol. I, pp. 19–99. Elsevier Science (2001)
2. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transaction on Computers 35(8), 677–691 (1986)
3. Fietzke, A., Weidenbach, C.: Labelled splitting. Ann. Math. Artif. Intell. 55(1-2), 3–34 (2009)
4. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (System description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
5. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 7, vol. I, pp. 371–443. Elsevier Science (2001)
6. Riazanov, A., Voronkov, A.: Splitting without backtracking. In: Nebel, B. (ed.) 17th International Joint Conference on Artificial Intelligence, IJCAI 2001, vol. 1, pp. 611–617 (2001)
7. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Commun. 15(2,3), 91–110 (2002)
8. Riazanov, A., Voronkov, A.: Limited resource strategy in resolution theorem proving. Journal of Symbolic Computations 36(1-2), 101–115 (2003)
9. Schulz, S.: E – a brainiac theorem prover. Journal of AI Communications 15(2-3), 111–126 (2002)
10. Sekar, R., Ramakrishnan, I.V., Voronkov, A.: Term indexing. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 26, vol. II, pp. 1853–1964. Elsevier Science (2001)
11. Sutcliffe, G.: The 4th ijcar automated theorem proving system competition - casc-j4. AI Communications 22(1), 59–72 (2009)
12. Sutcliffe, G.: The tptp problem library and associated infrastructure. J. Autom. Reasoning 43(4), 337–362 (2009)
13. Voronkov, A.: The anatomy of Vampire: Implementing bottom-up procedures with code trees. Journal of Automated Reasoning 15(2), 237–265 (1995)
14. Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 27, vol. II, pp. 1965–2013. Elsevier Science (2001)

Author Index