Pascal Fontaine
Christophe Ringeissen
Renate A. Schmidt (Eds.)

LNAI 8152

Frontiers of Combining Systems

9th International Symposium, FroCoS 2013
Nancy, France, September 2013
Proceedings

Springer

Lecture Notes in Artificial Intelligence 8152

Subseries of Lecture Notes in Computer Science

Pascal Fontaine Christophe Ringeissen
Renate A. Schmidt (Eds.)

Frontiers of Combining Systems

9th International Symposium, FroCoS 2013
Nancy, France, September 18-20, 2013
Proceedings

Springer

Volume Editors

Pascal Fontaine
LORIA, Inria Nancy Grand Est
Université de Lorraine
615 rue du Jardin Botanique
54602 Villers-les-Nancy, France
E-mail: pascal.fontaine@inria.fr

Christophe Ringeissen
LORIA, Inria Nancy Grand Est
615 rue du Jardin Botanique
54602 Villers-les-Nancy, France
E-mail: christophe.ringeissen@loria.fr

Renate A. Schmidt
The University of Manchester
School of Computer Science
Manchester M13 9PL, UK
E-mail: renate.schmidt@manchester.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-40884-7 e-ISBN 978-3-642-40885-4
DOI 10.1007/978-3-642-40885-4
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013947465

CR Subject Classification (1998): I.2.3, I.2, D.3.1, F.4, D.2.4, I.1, F.2

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 collects papers presented at the 9th International Symposium on Frontiers of Combining Systems (FroCoS 2013), held September 18–20, 2013, in Nancy, France. Previous FroCoS meetings were organized in Munich (1996), Amsterdam (1998), Nancy (2000), Santa Margherita Ligure (2002), Vienna (2005), Liverpool (2007), Trento (2009) and Saarbrücken (2011). In 2004, 2006, 2008, 2010, and 2012, FroCoS was a constituent of IJCAR, the International Joint Conference on Automated Reasoning. Like its predecessors, FroCoS 2013 offered a common forum for the presentation and discussion of research in the general area of combination, modularization, and integration of systems, with emphasis on logic-based systems and their practical use. This research touches on many areas of computer science such as logic, symbolic computation, program development and verification, artificial intelligence, knowledge representation, and automated reasoning.

This year FroCoS was co-located in Nancy with the 22nd International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2013) held September 16–19, 2013. This gave the opportunity to organize both joint scientific and joint social events.

The Program Committee accepted 20 papers out of a total of 33 submissions of overall high quality. In addition to the contributed papers, the program included four invited lectures:

- "Witness Runs for Counter Machines" by Stéphane Demri (LSV, CNRS & ENS de Cachan and New York University)
- "From Resolution and DPLL to Solving Arithmetic Constraints" by Konstantin Korovin (The University of Manchester)
- "Specification and Verification of Linear Dynamical Systems: Advances and Challenges" by Joël Ouaknine (Oxford University)
- "MetiTarski's Menagerie of Cooperating Systems" by Lawrence C. Paulson (University of Cambridge)

Stéphane Demri's presentation was a joint FroCoS-TABLEAUX invited lecture.

For the success of the conference, we want to thank several people and organizations. First, we would like to thank all authors who submitted papers and all participants of the conference for their contributions and presentations. We are grateful to the invited speakers not only for participating and their lectures, but also for contributing extended abstracts or full papers to the proceedings. We thank the members of the Program Committee and all the referees for the time and care spent on reviewing and selecting the papers. We thank the members of the FroCoS Steering Committee for their advice and support, and Andrei Voronkov for his EasyChair conference management system.

Special thanks go to the chairs of TABLEAUX 2013, Didier Galmiche and Dominique Larchey-Wendling, for the productive collaboration in organizing the co-location of FroCoS and TABLEAUX. Moreover, we are extremely grateful to the local team led by Anne-Lise Charbonnier and Louisa Touioui from the Manifestations Scientifiques service of Inria Nancy-Grand Est for the practical organization of the conference.

For institutional support, we thank Institut National de Recherche en Informatique et Automatique (Inria), Laboratoire Lorrain de Recherche en Informatique et ses Applications (LORIA), and the Formal Methods Department of LORIA, Centre National de la Recherche Scientifique (CNRS), the Université de Lorraine, the Communauté Urbaine du Grand Nancy and the Région Lorraine.

July 2013

Pascal Fontaine
Christophe Ringeissen
Renate Schmidt

Conference Organization

Program Chairs

Pascal Fontaine	LORIA, Inria Nancy-Grand Est, Université de Lorraine, France
Renate Schmidt	The University of Manchester, UK

Conference Chair

Christophe Ringeissen	LORIA, Inria Nancy-Grand Est, France

Program Committee

Carlos Areces	Universidad Nacional de Córdoba, Argentina
Alessandro Artale	Libera Università Bolzano, Italy
Franz Baader	Technische Universität Dresden, Germany
Clark Barrett	New York University, USA
Peter Baumgartner	NICTA, Canberra, Australia
Christoph Benzmüller	Freie Universität Berlin, Germany
Jasmin Christian Blanchette	Technische Universität München, Germany
Thomas Bolander	Danmarks Tekniske Universitet, Denmark
Clare Dixon	University of Liverpool, UK
François Fages	Inria Paris-Rocquencourt, France
Pascal Fontaine	LORIA, Inria Nancy-Grand Est, Université de Lorraine, France
Didier Galmiche	LORIA, Université de Lorraine, France
Vijay Ganesh	University of Waterloo, Canada
Silvio Ghilardi	Università degli Studi di Milano, Italy
Guido Governatori	NICTA, Queensland, Australia
Bernhard Gramlich	Technische Universität Wien, Austria
Katsumi Inoue	National Institute of Informatics, Japan
Sava Krstić	Intel Corporation, USA
Alessio Lomuscio	Imperial College London, UK
Till Mossakowski	Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Bremen, Germany
Silvio Ranise	Fondazione Bruno Kessler, Trento, Italy
Christophe Ringeissen	LORIA, Inria Nancy-Grand Est, France
Philipp Rümmer	Uppsala Universitet, Sweden
Renate Schmidt	The University of Manchester, UK
Roberto Sebastiani	Università degli Studi di Trento, Italy
Viorica Sofronie-Stokkermans	Universität Koblenz-Landau, Germany

Andrzej Szałas	Linköpings Universitet, Sweden, Uniwersytetu Warszawskiego, Poland
René Thiemann	Universität Innsbruck, Austria
Ashish Tiwari	SRI International, USA
Josef Urban	Radboud Universiteit, The Netherlands
Christoph Weidenbach	Max-Planck-Institut für Informatik, Germany
Frank Wolter	University of Liverpool, UK

External Reviewers

Yohan Boichut
Guillaume Burel
Horatiu Cirstea
Bernardo Cuenca Grau
Stephanie Delaune
Morgan Deters
Alastair Donaldson
Stephan Falke
Arnaud Fietzke
Xiang Fu
Carsten Fuhs
Klaus Frovin Joergensen
Jean Christoph Jung
Miyuki Koshimura
Peter Lammich
Dominique Larchey-Wendling
Vladimir Lifschitz
Michel Ludwig
Morgan Magnin
Pierre Marquis
Thierry Martinez
Aart Middeldorp
Barbara Morawska
Jan Otop
Gian Luca Pozzato
Florian Rabe
Martin Rezk
Agnieszka Rusinowska
Vladislav Ryzhikov
Peter Schneider-Kamp
Ilya Shapirovsky
Martin Suda
Michele Vescovi
Jonathan von Schroeder
Freek Wiedijk
Thomas Wies

Sponsoring Institutions

Institut National de Recherche en Informatique et Automatique (Inria)
Laboratoire Lorrain de Recherche en Informatique et ses Applications (LORIA)
Formal Methods Department, LORIA
Centre National de la Recherche Scientifique (CNRS)
Université de Lorraine
Communauté Urbaine du Grand Nancy
Région Lorraine

Table of Contents

Invited Talk 1

Inductive Theorem Proving

Arrays and Memory Access Optimization

Approximation and Forgetting

Invited Talk 2

Temporal and Description Logic Techniques

Invited Talk 3

Theorem Proving with Theories and Sorts

Invited Talk 4

Modal Logic and Description Logic

Rewriting

MetiTarski's Menagerie of Cooperating Systems

Lawrence C. Paulson

Computer Laboratory, University of Cambridge, England
lp15@cl.cam.ac.uk

Abstract. MetiTarski, an automatic theorem prover for real-valued special functions, is briefly introduced. Its architecture is sketched, with a focus on the arithmetic reasoning systems that it invokes. Finally, the paper describes some applications where MetiTarski is itself invoked by other tools.

1 Introduction

As we all know, connecting systems together is easy; the difficulty lies in getting them to cooperate productively. Combining theorem proving with computer algebra has long been regarded as a promising idea, but it has been difficult to realise in practice. MetiTarski is an automatic theorem prover for real-valued special functions [2]. In its original form it consisted of two separate systems linked together: Metis [14,15] (a resolution theorem prover) and QEPCAD [5,13] (a quantifier elimination procedure for real-closed fields). Today, MetiTarski can invoke three separate reasoning tools (QEPCAD, Mathematica and Z3) and can itself be invoked by other tools, in particular, KeYmaera and PVS.

2 Architectural Overview

The core idea in MetiTarski is to reduce problems involving special functions (sin, cos, ln, etc.) to decidable polynomial inequalities, which can then be supplied to QEPCAD. First-order formulas over polynomial inequalities over the real numbers admit quantifier elimination [11], and are therefore decidable. This decision problem is known as RCF, for real closed fields. Dolzmann et al. [10] have written a useful overview of both the theory and its practical applications.

An early design decision was to adopt an existing theorem prover (namely Metis), rather than to write a tableau-style theorem prover from scratch, which was the approach adopted for Analytica [7] and Weierstrass [3], two earlier systems that combined mathematical software with logic. It seemed clear to us that the resolution method would turn out to be much more sophisticated and effective than the naive methods our small group would be able to concoct on our own. Instead of having to write an entire theorem prover, we would merely need to write some interface code and modify certain standard aspects of resolution. Arithmetic simplification obviously had to be introduced (for example, to identify $2x + y$ with $x + y + 0 + x$), and the standard mechanisms for selecting

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 1–6, 2013.

the most promising clause and literal were tuned to our application [1,2]. Early versions of MetiTarski performed well despite having only a modest amount of specialist code. By now, however, we have extended MetiTarski's code base extensively. We introduced case-splitting with backtracking [4], as is found in SMT solvers. We also included our own code for interval constraint solving, to either supplement or replace the external decision procedures.

MetiTarski relies on collections of upper and lower bounds (consisting of polynomials or rational functions) for the various special functions. The main effort in 2009 focused on refining these bounds, in particular through the introduction of continued fractions. Resolution chooses which axioms to use in a proof automatically. A single proof may use different axioms to cover different intervals of the region under consideration. A further benefit of our use of standard resolution is that other forms of axioms (concerning the absolute value function, or the min and max functions) can be written in first-order logic. The absolute value axioms state the obvious properties:

$$\neg(0 \leq x) \vee |x| = x \qquad 0 \leq x \vee |x| = -x$$

Resolution performs the appropriate sign reasoning automatically.

Resolution operates on *clauses*, or disjunctions of *literals*, which for MetiTarski are typically real inequalities. Under certain circumstances, MetiTarski can simplify the selected disjunction by formulating a problem that it can submit to an external decision procedure. Such problems involve a particular inequality in a disjunction, within its context. This context consists of the remainder of the disjunction and certain global facts. If the decision procedure finds the conjunction of these assertions to be inconsistent, then the inequality can be deleted from the clause. This connection between the basic resolution method and an external decision procedure is the key idea. Given another application domain, other decision procedures could probably be substituted for those called by MetiTarski. The only difficulty is that such a system would probably have to compete head-on with SMT solvers, which are highly refined and effective.

MetiTarski also uses the decision procedure for a form of redundancy elimination. As the proof search proceeds, polynomial formulas accumulate, and these are supplied to every decision procedure call. But if some of these formulas are redundant, they slow down subsequent calls without providing any benefit. Therefore, every time a new polynomial formula emerges from the resolution process, it is tested for redundancy—does it follow in the theory of RCF from previously known formulas?—and possibly discarded. The poor complexity of RCF quantifier elimination makes this step necessary.

3 MetiTarski's Decision Procedures

We adopted QEPCAD originally because it was free and easy to use, dedicated as it was to the single task of quantifier elimination. Moreover, QEPCAD worked extremely well in our first experiments. But QEPCAD had a number of limitations, concerning both portability (the code base seems to date from the distant past) and performance. Our decision problem is inherently intractable: doubly exponential in the number of variables in the problem [8]. This caused no difficulties at first, when virtually our entire problem set was univariate, but there are other ways to settle univariate special-function inequalities, and many important problems involve multiple variables.

Mathematica, the well-known computer algebra system, was next to be integrated with MetiTarski, as an alternative to QEPCAD. Though we regret the reliance on commercial software, many institutions already have Mathematica licences, and its quantifier elimination procedure is much more modern and powerful than QEPCAD's. It copes with problems in up to five variables, where QEPCAD cannot be expected to terminate at all. Mathematica has many configurable options, leaving us with many possible refinements to investigate. Mathematica can solve many special-function inequalities itself, and MetiTarski can take advantage of this capability to solve even harder problems.

The theorem prover Z3 [9], with its new extension for non-linear arithmetic [17], provides the third of our decision procedures. The great advantage for us is the possibility of working with its developers. We can tune it to our specific needs. Where it performs badly, we can send the problems for examination and know that they will be looked at. In some cases, Z3 has coped with problems in up to nine variables [21]. Z3 is free to non-commercial users.

Much of the effort needed to integrate different systems concerns overcoming conceptually trivial but serious obstacles. For many months, our team struggled with mysterious failures involving QEPCAD. These mainly happened during lengthy, overnight regression testing, where certain jobs would mysteriously hang and eventually bring all testing to a halt. Eventually, the problem was isolated to one of QEPCAD's peculiarities: unless it is used at a normal terminal, it performs its own echoing of input lines. (This allowed it to produce a readable output transcript when running in batch mode.) Because the inputs to QEPCAD can exceed 50K characters, and because MetiTarski never reads the output of QEPCAD until after it has sent a full problem to it, QEPCAD's output buffer would fill up, blocking its execution. Similar difficulties involving the other decision procedures take a surprising amount of time to diagnose and fix. Today as I write this, we are struggling with a mysterious problem plaguing integration with Z3.

Note that the choice among these three decision procedures is not straightforward. QEPCAD performs best in many situations.

4 Ongoing Research

We can often get better results if we do not regard the reasoning components of our system as black boxes. Automatically generated problems tend to be regular, and should if possible be tailored to the strengths of the component that will process them, or conversely, that component could itself be modified to perform better on those automatically generated problems. In the case of Z3, we were able to find a number of refinements that greatly improved its performance with MetiTarski [20]. One such refinement is to switch off a processing stage (univariate polynomial factorisation) that we could predict to be unnecessary. Another refinement, called *model sharing*, involved Z3 passing counterexamples to MetiTarski that it could use to eliminate some future Z3 calls.

Choosing which of the arithmetic solvers to call, given a particular problem, is itself a research question. A Cambridge student, Zongyan Huang, is currently investigating whether machine learning can be effective here. The basic idea is that features present in the special-function problem originally given to MetiTarski may be sufficient to predict which decision procedure will perform best on the polynomial decision problems that MetiTarski will generate for that problem. Features that we are examining include which special functions are present and how many variables there are. Zongyan is using Support Vector Machines (SVMs). This modelling approach, implemented as SVM-Light [16], is a form of machine learning that offers good results with reasonable efficiency. Her work is still experimental, but if it is successful, then realising it would involve MetiTarski running some machine learning code near the beginning of its execution.

MetiTarski opens the possibility of verifying dynamical systems using nonlinear models involving transcendental functions. Such models are common in engineering, for example in problems involving rotation. William Denman is investigating this area. He uses Mathematica (manually) to derive differential equations to model a given dynamical system. Such a model is a system of differential equations. Denman has written a Python program based on the algorithm implemented in HybridSAL [23], which is a tool for creating discrete models of hybrid systems. His program transforms the system of differential equations into a set of MetiTarski problems. MetiTarski is used to identify infeasible states in the abstract model, thereby simplifying it; the attraction of this approach is that it does not require MetiTarski to solve all the problems. The outcome of this procedure is a discrete, finite model suitable for model checking (currently, using NuSMV [6]).

5 Prospects for Further Integration

KeYmaera is a sophisticated interactive theorem prover designed for verifying hybrid systems [22]. We have recently joined MetiTarski to KeYmaera as a backend, hoping to provide the possibility of verifying systems whose models involve special functions. PVS is an interactive theorem prover designed for a variety of application areas, including hardware and hybrid systems [19]. William Denman, in collaboration with César Muñoz, has created an experimental linkup between MetiTarski and PVS. In both cases, the calling system invokes MetiTarski and trusts the result. These experiments should help identify new application areas for MetiTarski, suggesting areas for further development as well as providing justification for the effort needed to build a more robust integration. MetiTarski returns machine-readable proofs that combine standard resolution steps with a few additional inference rules, reflecting its use of computer algebra computations steps. These proofs can be used to facilitate the integration of MetiTarski with other systems, even if MetiTarski's conclusions are not trusted. In such applications, MetiTarski becomes a hub lying at the centre of a network of communicating reasoners.

The motivation for this research, years ago, was to equip Isabelle (an interactive theorem prover [18]) with support for reasoning about special functions. The original idea was to use lightweight methods that could prove relatively easy theorems. MetiTarski can prove difficult theorems, but through heavyweight methods that are difficult to include in an LCF-style theorem prover such as Isabelle. In such theorem provers, there is a strong preference to use only tools that justify every step in the proof kernel; so-called oracles that trust an external reasoner are frowned upon. The PVS community is more accommodating to oracles, and the present linkup between PVS and MetiTarski will be invaluable for investigating the potential of such combined systems. An integration with Sage [12], an open-source computer algebra system, is also planned for the near future.

Acknowledgements. Other members of the team include (in Edinburgh) Paul Jackson, Grant Passmore and Andrew Sogokon, and (in Cambridge) James Bridge, William Denman and Zongyan Huang. In the text above, "we" refers to all of us. We are grateful to our outside collaborators such as Eva Navarro, André Platzer, and many others not listed here.

The research was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/C013409/1, EP/I011005/1, EP/I010335/1].

References

1. Akbarpour, B., Paulson, L.C.: MetiTarski: An automatic prover for the elementary functions. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC/Calculemus/MKM 2008. LNCS (LNAI), vol. 5144, pp. 217–231. Springer, Heidelberg (2008)
2. Akbarpour, B., Paulson, L.: MetiTarski: An automatic theorem prover for real-valued special functions. Journal of Automated Reasoning 44(3), 175–205 (2010)
3. Beeson, M.: Automatic generation of a proof of the irrationality of e. Journal of Symbolic Computation 32(4), 333–349 (2001)
4. Bridge, J., Paulson, L.: Case splitting in an automatic theorem prover for real-valued special functions. Journal of Automated Reasoning (2012) (in press), `http://dx.doi.org/10.1007/s10817-012-9245-6`
5. Brown, C.W.: QEPCAD B: a program for computing with semi-algebraic sets using CADs. SIGSAM Bulletin 37(4), 97–108 (2003)
6. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An openSource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
7. Clarke, E., Zhao, X.: Analytica: A theorem prover for Mathematica. Mathematica Journal 3(1), 56–71 (1993)
8. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. J. Symbolic Comp. 5, 29–35 (1988)

9. 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)
10. Dolzmann, A., Sturm, T., Weispfenning, V.: Real quantifier elimination in practice. In: Heinrich Matzat, B., Greuel, G.-M., Hiss, G. (eds.) Algorithmic Algebra and Number Theory, pp. 221–247. Springer (1999)
11. van den Dries, L.: Alfred Tarski's elimination theory for real closed fields. Journal of Symbolic Logic 53(1), 7–19 (1988)
12. Gray, M.A.: Sage: A new mathematics software system. Computing in Science Engineering 10(6), 72–75 (2008)
13. Hong, H.: QEPCAD — quantifier elimination by partial cylindrical algebraic decomposition, Sources and documentation are on the Internet at `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`
14. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Archer, M., Di Vito, B., Muñoz, C. (eds) Design and Application of Strategies/Tactics in Higher Order Logics, NASA/CP-2003-212448 in NASA Technical Reports, pp. 56–68 (September 2003)
15. Hurd, J.: Metis first order prover (2007), Website at `http://gilith.com/software/metis/`
16. Joachims, T.: Making large-scale support vector machine learning practical. In: Schölkopf, B., Burges, C.J.C., Smola, A.J. (eds.) Advances in Kernel Methods, pp. 169–184. MIT Press (1999)
17. 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)
18. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
19. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.K.: PVS: Combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996)
20. Passmore, G.O., Paulson, L.C., de Moura, L.: Real algebraic strategies for MetiTarski proofs. In: Jeuring, J., Campbell, J.A., Carette, J., Dos Reis, G., Sojka, P., Wenzel, M., Sorge, V. (eds.) CICM 2012. LNCS, vol. 7362, pp. 358–370. Springer, Heidelberg (2012)
21. Paulson, L.C.: MetiTarski: Past and future. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 1–10. Springer, Heidelberg (2012)
22. Platzer, A., Quesel, J.-D.: KeYmaera: A hybrid theorem prover for hybrid systems (System description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 171–178. Springer, Heidelberg (2008)
23. Tiwari, A.: HybridSAL relational abstracter. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 725–731. Springer, Heidelberg (2012)

Combining Superposition and Induction: A Practical Realization[*]

Abdelkader Kersani and Nicolas Peltier

University of Grenoble (LIG, CNRS)

Abstract. We consider a proof procedure aiming at refuting clause sets containing arithmetic constants (or parameters), interpreted as natural numbers. The superposition calculus is enriched with a loop detection rule encoding a form of mathematical induction on the natural numbers (by "descente infinie"). This calculus and its theoretical properties are described in [2,16]. In the present paper, we focus on more practical aspects. We provide algorithms to apply the loop detection rule in an automatic and efficient way. We describe a research prototype implementing our technique and provide some preliminary experimental results.

1 Introduction

We consider first-order formulæ built on a language containing constant symbols interpreted as natural numbers. As an example, consider the formula ϕ defined as the conjunction of the following formulæ:

$$\begin{gathered} p(0,a) \\ \forall x, y\, \neg p(x,y) \vee p(x+1, f(y)) \\ \exists n \forall x\, \neg p(n,x) \end{gathered}$$

The formula ϕ is satisfiable in the usual sense, but it is unsatisfiable if the sort of the first argument of p is interpreted as the natural numbers (with the usual interpretation of $0, 1$ and $+$). Then the existential variable n must be interpreted as a natural number k, and it is easy to check, by induction on k, that the first two formulæ entail that $p(k, f^k(a))$ holds, which implies that the formula is unsatisfiable. Existing resolution or superposition based theorem provers cannot establish the unsatisfiability of such formulæ since they are based on standard first-order logic. Proof procedures (based on several different approaches) have been proposed to handle hybrid formulæ, mixing first-order logic with interpreted theories such as Presburger arithmetic [4,1,6,12] but they do not handle inductive theorems. When fed with the previous formula, these approaches will infer the infinite set of formulæ $n \not\simeq 0$, $n \not\simeq 1$, $n \not\simeq 2$, ... (where n denotes the Skolem constant derived from the quantification $\exists n$), but will not detect unsatisfiability in finite time (since Presburger arithmetic is not compact). The standard approach

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

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 7–22, 2013.

for dealing with inductive theorems in the context of first-order theorem proving is to add explicit induction schemes. For instance, in the previous case, one can replace the formula $\exists n \forall x \, \neg p(n,x)$ by $\exists n \forall x \, \neg p(n,x) \wedge \forall m \, m+1 \not\simeq n \vee \exists x \, p(m,x)$ (stating that $\forall x \, \neg p(m,x)$ holds for $m = n$ but not for the predecessor of n) which can be easily derived by assuming that n is the *minimal* natural number satisfying the property $\forall x \, \neg p(n,x)$. Alternatively, one can also add the usual induction scheme using $\exists x \, p(m,x)$ as an inductive invariant:

$$(\exists x \, p(0,x) \wedge \forall m((\exists x \, p(m,x)) \Rightarrow (\exists x \, p(m+1,x)))) \Rightarrow \forall m \exists x \, p(m,x)$$

Using these additional axioms, the unsatisfiability of ϕ can easily be established by any theorem prover. The inductive rule defined in [15] also relies on the use of explicit induction schemes.

However, this approach relies on the user to guess the right inductive lemma. The inductive invariant is not necessarily equivalent to the goal, and is not even bound to occur in the initial formula (it is well-known that inductive proofs do not admit cut elimination). For instance, if the third formula is replaced by: $\exists n \forall x \, \neg q(n,x)$ with the additional axiom: $\forall x, y \, q(x,y) \vee \neg p(x,y)$, then the formula cannot be established by using the negation of the goal $\exists x \, q(n,x)$ as an inductive invariant: one has to use $\exists x \, p(n,x)$ instead.

Another approach consists in using inductive theorem provers, which are usually based on rewriting [7,13,14,20,11]. These approaches allow one to generate automatically the induction lemmata (in some cases). Intuitively, these procedures work as follows: the goal is rewritten using axioms until it can be reduced to `true` or `false`. Of course only the ground instances of the goal can be normalized and enumerating those instances does not terminate in general. In order to ensure termination in some cases, the previously encountered goals can themselves be used as derived rewrite rules, provided the considered terms are strictly lower than the initial ones, according to some reduction ordering, which can be either fixed a priori or constructed dynamically all along the search. This technique allows one to simulate the application of inductive hypotheses without having to state explicitly the inductive invariants (of course additional inductive lemmata still have to be added by hand in many cases). However, these approaches are restricted to goals of the form $\forall \boldsymbol{x} \, \psi$ where ψ is a quantifier-free formulæ, thus they cannot handle formulæ as ϕ in the previous example, whose goal is of the form $\forall n \exists x \, \psi$, before negation (the inductive theorem prover SPIKE has been extended in order to handle existential variables [5], but the use of such variables is strongly restricted). The "inductionless induction" approach [8], which reduces inductive theorem proving to a consistency test in first-order logic, suffers from the same limitation.

In previous work [2,16], we have presented an extension of the superposition calculus which is tailored to handle formulæ such as the previous one. The idea is twofold. First, the arithmetic terms are abstracted away and replaced by variables, in order to allow inferences on them. This allows one to get rid of first-order symbols in order to derive properties of pure arithmetic terms. Second, the usual inference rules of the calculus are enriched with a new rule allowing to detect cycles in the derivations. These loops correspond to the inductive invariants that are needed to establish the validity of the theorem. A first definition of the loop detection rule is given in [2] and a more general version is provided in [16] yielding stronger completeness results (of course the method is not complete in general, since the logic is not even semi-decidable [16]). Roughly speaking, these rules apply when a set of clauses $S[n]$ is generated (where n is an arithmetic constant) such that the set $n \geq k \wedge S[n-k]$ can be derived from $S[n]$ (using the inference rules of the calculus). By descente infinie, it is clear that this implies that S is unsatisfiable. The soundness of this approach is proven in [2,16], and some (partial) completeness results are presented. In the present paper, we tackle more practical aspects, namely the efficient generation of the sets of clauses on which the loop detection rule can be applied. The problem consists in finding efficiently sets of clauses S satisfying the relation above. We present two different algorithms for performing this task (each with their pros and cons). We describe an implementation of our method and provide some preliminary experimental results.

2 Syntax and Semantics

We firstly define the syntax and semantics of the considered logic. We assume some familiarity with the usual notions in logic and automated deduction (missing definitions can be found in, e.g., [18]). We consider two distinct sorts: the sort `term` of the standard terms, and the sort `nat` of the natural numbers. The set of terms is built as usual on a set of function symbols Σ and on a set of variables $\mathcal{X}$. The signature Σ contains in particular the symbols 0 and *succ*, of profile `nat` and `nat` $\rightarrow$ `nat` respectively. We assume that Σ contains no other symbol of range `nat`.

An *atom* is an equation of the form $t \simeq s$ where t and s are terms of sort `term`. A *literal* is either an atom (positive literal) or the negation of an atom (negative literal). A *clause* is a finite set (written as a disjunction) of literals. Let n be a special symbol, called the *parameter*, not occurring in Σ (n is intended to denote a natural number, and can be viewed as a constant symbol of sort `nat`).

Definition 1. *An n-*clause *is a pair $[C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ where C is a clause and the t_i's $(1 \leq i \leq k)$ are terms of sort* `nat`*. It is* normalized *if $k \in \{0,1\}$. If $k = 0$, $\bigwedge_{i=1}^{k} n \simeq t_i$ is equivalent to* `true` *by convention and $[C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ is simply written C. C is the* clausal part *of the n-clause, and $\bigwedge_{i=1}^{k} n \simeq t_i$ is the* constraint*.*

Note that by definition, n can only occur in the constraint part of the n-clause (since $n \notin \Sigma$). Thus an expression of the form $f(n) \simeq a$ for instance is to be written as $[f(x) \simeq a \mid n \simeq x]$, where x is a variable of sort `nat`.

For every expression e, $\mathrm{var}(e)$ denotes the set of variables occurring in e. An expression is *ground* if it contains no variable.

A *substitution* σ is a function mapping every variable to a term of the same sort. The *domain* of σ is the set of variables x such that $x\sigma \neq x$. For every expression e, $e\sigma$ denotes as usual the expression obtained from e by replacing every occurrence of each variable x by $x\sigma$. The substitution σ is *ground* iff for every variable x in the domain of σ, $x\sigma$ is ground.

The terms $t_1, \dots, t_k$ are *unifiable* iff there exists a substitution σ such that $t_1\sigma = \dots = t_k\sigma$. Any set of unifiable terms has a most general unifier (unique up to a renaming).

We identify a term $succ^k(0)$ with the natural number k; thus we write, e.g., $succ^k(0) < succ^l(0)$ for $k < l$, or $k + t$ for $succ^k(t)$.

Interpretations are usually defined as congruences on the set of terms. In our setting, we also have to specify the value of the symbol n (the symbols 0 and *succ* are interpreted as free constructors, note that the clauses contain no equations between natural numbers). This yields the following:

Definition 2. *An* interpretation $\mathcal{I}$ *is defined by a pair* $(n_{\mathcal{I}}, \simeq_{\mathcal{I}})$*, where* $n_{\mathcal{I}}$ *is natural number (i.e., a term of the form* $succ^k(0)$*) and* $\simeq_{\mathcal{I}}$ *is a congruence on the set of ground terms of sort* `term`.

The notion of validity is defined in a very natural way:

Definition 3. *An interpretation* $\mathcal{I}$ validates *an expression* E *(written* $\mathcal{I} \models E$*) iff one of the following conditions holds.*

- *E is a ground literal $t \simeq s$ (resp. $t \not\simeq s$) and $t \simeq_{\mathcal{I}} s$ (resp. $t \not\simeq_{\mathcal{I}} s$).*
- *E is a ground clause $\bigvee_{i=1}^{k} l_i$ and there exists $i \in [1, k]$ such that $\mathcal{I} \models l_i$.*
- *E is an n-clause $[C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ and for every ground substitution σ of domain $var(E)$ such that $\forall i \in [1, k], n_{\mathcal{I}} = t_i\sigma$, it holds that $\mathcal{I} \models C\sigma$.*
- *E is a set of n-clauses and $\forall \mathcal{C} \in E, \mathcal{I} \models \mathcal{C}$.*

An interpretation validating E is a model *of E. We write $E \models E'$ if every model of E is a model E'. Two expressions E and E' are* equivalent *(written $E \equiv E'$) if $E \models E'$ and $E' \models E$. A* tautology *is an expression equivalent to* **`true`**.

By definition, $\mathcal{I} \models [\Box \mid n \simeq succ^k(0)]$ iff $n_{\mathcal{I}} \neq k$. Similarly, $\mathcal{I} \models [\Box \mid n \simeq succ^k(x)]$ iff $n_{\mathcal{I}} < k$ (where x is a variable). Consequently, an n-clause of the form $[\Box \mid n \simeq succ^k(0)]$ (resp. $[\Box \mid n \simeq succ^k(x)]$) will be written $n \not\simeq k$ (resp. $n < k$).

If $\mathcal{I}$ is an interpretation and k is a natural number, we denote by $\mathcal{I}[k/n]$ the interpretation coinciding with $\mathcal{I}$, except that the value of n is set to k.

The following proposition shows that every non-tautological n-clause is equivalent to a normalized n-clause.

Proposition 1. *Let $\mathcal{C} = [C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ be an n-clause. If $t_1, \dots, t_n$ are unifiable, then $\mathcal{C}$ is equivalent to $[C\sigma \mid n \simeq t_1\sigma]$, where σ is an m.g.u. of $t_1, \dots, t_n$. Otherwise $\mathcal{C}$ is a tautology.*

Thanks to Proposition 1 we can safely assume that every n-clause is normalized (the normalization operation is applied in a systematic way to every generated n-clause).

The usual relation of subsumption extends straightforwardly to n-clauses:

Definition 4. *Let* $\mathcal{C} = [C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ *and* $\mathcal{C}' = [C' \mid \bigwedge_{i=1}^{l} n \simeq t'_i]$ *be two* n*-clauses. The* n*-clause* $\mathcal{C}$ subsumes $\mathcal{C}'$ *(written* $\mathcal{C} \leq_{sub} \mathcal{C}'$*) if there exists a substitution* σ *such that* $C\sigma \subseteq C'$ *and* $\{t_1, \ldots, t_k\}\sigma \subseteq \{t'_1, \ldots, t'_l\}\sigma$.

Proposition 2. *If* $\mathcal{C} \leq_{sub} \mathcal{C}'$ *then* $\mathcal{C} \models \mathcal{C}'$.

The subsumption relation $\leq_{sub}$ can be extended to sets of n-clauses: we write $S \leq_{sub} S'$ if for every $\mathcal{C}' \in S'$, there exists $\mathcal{C} \in S$ such that $\mathcal{C} \leq_{sub} \mathcal{C}'$.

By Definition 3, an n-clause $[C \mid n \simeq succ^i(x)]$ (with $x \in \mathcal{X}$) is equivalent to an expression of the form $n \geq i \Rightarrow C[n-i]$. The *rank* of $[C \mid n \simeq succ^i(x)]$ is the number r such that $n - r$ is the maximal expression containing n occurring in $C[n-i]$. For instance, consider the n-clause $[f(x, succ(y)) \simeq y \mid n \simeq succ^i(x)]$. It is equivalent to the expression $f(x, succ(n-i)) \simeq n-i$, i.e., $f(x, n-(i-1)) \simeq n-i$, hence its rank is $i - 1$. Note that if C contains no occurrence of *succ* then the rank of $[C \mid n \simeq succ^i(x)]$ is simply i. For every set of n-clauses S and for every natural number i, we denote by $S[i]$ the set of n-clauses of rank i in S. We denote by $S[\top]$ the set of n-clauses whose constraint is `true`.

3 Superposition Calculus

The usual superposition calculus can easily be extended to operate on n-clauses. Let $<$ be a reduction ordering and let *sel* be a selection function, mapping every clause C to a subset of C, such that either *sel*(C) contains a negative literal, or *sel*(C) contains all the $<$-maximal literals in C. The calculus is defined by the following three rules (the premisses are assumed to be normalized). As usual $t|_p$ is the term occurring at position p in t, and $t[s]_p$ is the term obtained from t by replacing the subterm at position p by s.

Superposition

$$\frac{[C \vee t \bowtie s \mid \mathcal{X}], [D \vee u \simeq v \mid \mathcal{Y}]}{[C \vee D \vee t[v]_p \bowtie s \mid \mathcal{X} \wedge \mathcal{Y}]\sigma}$$

If $\bowtie \in \{\simeq, \not\simeq\}$, $\sigma = \text{mgu}(u, t|_p)$, $u\sigma \not\leq v\sigma, t\sigma \not\leq s\sigma$, $t|_p$ is not a variable, $(t \bowtie s)\sigma \in sel((C \vee t \bowtie s)\sigma)$, $(u \simeq v)\sigma \in sel((D \vee u \simeq v)\sigma)$.

Reflection

$$\frac{[C \vee t \not\simeq s \mid \mathcal{X}]}{[C \mid \mathcal{X}]\sigma} \qquad \text{If } \sigma = \text{mgu}(t, s),\ (t \not\simeq s)\sigma \in sel((C \vee t \not\simeq s)\sigma)$$

Factorisation

$$\frac{[C \vee t \simeq s \vee u \simeq v \mid \mathcal{X}]}{[C \vee s \not\simeq v \vee t \simeq s \mid \mathcal{X}]\sigma}$$

If $\sigma = \text{mgu}(t, u)$, $t\sigma \not< s\sigma$, $u\sigma \not< v\sigma$, $(t \simeq s)\sigma \in sel((C \vee t \simeq s \vee u \simeq v)\sigma)$.

Example 1. The second example of the Introduction is formalized as follows.

1 $p(0, a) \simeq \mathtt{true}$
2 $p(x, y) \not\simeq \mathtt{true} \vee p(succ(x), f(y)) \simeq \mathtt{true}$
3 $[q(x, y) \not\simeq \mathtt{true} \mid n \simeq x]$
4 $q(x, y) \simeq \mathtt{true} \vee p(x, y) \not\simeq \mathtt{true}$

The following n-clauses are generated (for the sake of clarity, the literals of the form $\mathtt{true} \not\simeq \mathtt{true}$ are removed from the clauses):

5	$[p(x, y) \not\simeq \mathtt{true} \mid n \simeq x]$	(superposition, 4, 3)
6	$[\Box \mid n \simeq 0]$	(superposition, 1, 5)
7	$[p(x, y) \not\simeq \mathtt{true} \mid n \simeq succ(x)]$	(superposition, 2, 5)
8	$[\Box \mid n \simeq succ(0)]$	(superposition, 1, 7)
9	$[p(x, y) \not\simeq \mathtt{true} \mid n \simeq succ(succ(x))]$	(superposition, 2, 7)

An infinite number of n-clauses of the form $[\Box \mid n \simeq succ^k(0)]$ (i.e. $n \not\simeq k$) can be generated.

4 Loop Detection

In this section, we define the key part of the proof procedure, namely the loop detection rule. We provide a simpler and abstract version of the rule (compared with those in [2,16]) which is sufficient for our purposes (the loop detection rule provided in [16] is much more general because it handles parameters interpreted as words instead of natural numbers). In the next section, we will introduce new definitions and algorithms allowing for an efficient application of the rule. We first need to introduce some notations.

Definition 5. *For every natural number i and for every n-clause $\mathcal{C} = [C \mid \bigwedge_{j=1}^{k} n \simeq t_j]$, we denote by $\mathcal{C}{\downarrow_i}$ the n-clause $[C \mid \bigwedge_{j=1}^{k} n \simeq succ^i(t_j)]$. If S is a set of n-clauses, then $S{\downarrow_i} \overset{def}{=} \{\mathcal{C}{\downarrow_i} \mid \mathcal{C} \in S\}$.*

Intuitively, $S{\downarrow_i}$ denotes the same formula as S, with n replaced by $n - i$. This yields the following:

Proposition 3. *Let $i, j \in \mathbb{N}$, let S be a set of n-clauses and let $\mathcal{I}$ be an interpretation. If $\mathcal{I} \models S{\downarrow_j}$ and $\mathcal{I}(n) = i + j$ then $\mathcal{I}[i/n] \models S$.*

Proof. Let $\mathcal{C} = [C \mid \bigwedge_{l=1}^{k} n \simeq t_l] \in S$. Let σ be a substitution such that $\forall l \in [1, k]\, n_{\mathcal{I}[i/n]} = t_i\sigma$. This implies that $\forall l \in [1, k], t_l\sigma = i$ since $n_{\mathcal{I}[i/n]} = i$ by definition. We have $\mathcal{C}{\downarrow_j} = [C \mid \bigwedge_{l=1}^{k} n \simeq succ^j(t_l)]$. Since $t_l\sigma = i$ and $n_{\mathcal{I}} = i + j$ we have $n_{\mathcal{I}} = t_l\sigma$, for all $l \in [1, k]$. Since $\mathcal{I} \models S{\downarrow_j}$, this entails that $\mathcal{I} \models C\sigma$, and thus $\mathcal{I}[i/n] \models C\sigma$ (since $C\sigma$ contains no occurrence of n).

Definition 6. *Let S be a set of clauses. A pair of natural numbers (i, j) (with $j \neq 0$) is an* inductive loop *for S if there exists a set $S' \subseteq S$ such that $S' \models n \not\simeq l$, for every $l \in [i, i + j[$ and $S' \models S'{\downarrow_j}$*

Theorem 1. *If (i, j) is an inductive loop for S, then $S \models n < i$.*

Proof. We have $\mathcal{I} \models S'$. Let k be the minimal natural number greater or equal to i such that S' has a model $\mathcal{I}$ with $n_{\mathcal{I}} = k$. If $k < i+j$, then we have $S' \models n \not\simeq k$, which is impossible since $\mathcal{I} \not\models n \not\simeq k$, by definition. Otherwise, we have $\mathcal{I} \models S'{\downarrow_j}$, hence by Proposition 3 we deduce that $\mathcal{I}[k-j/n] \models S'$, which is impossible by minimality of k since $i \leq k - j < k$.

5 Practical Application of the Loop Detection Rule

This section contains the main new results of the paper. We define algorithms to test whether a pair of natural numbers (i, j) is an inductive loop. To this purpose, we have to check whether there exists a set of n-clauses S' satisfying the conditions of Definition 6. In practice, these conditions cannot be tested since semantic entailment is undecidable. We will thus only check whether the formulæ $n \not\simeq l$ (with $i \leq l < i+j$) and $S'{\downarrow_j}$ can be derived from S' using inferences previously performed by the prover. Furthermore, we will impose the additional restriction that all the n-clauses in S' containing n occur in the set $S[i]$. This condition greatly decreases the search space and it preserves the completeness results in [2,16].

We proceed in two steps. First we transform the semantic conditions of Definition 6 into purely syntactic properties and then we provide algorithms to test these properties in an effective way. We need to introduce additional notations.

Definition 7. *Let S be a set of n-clauses. An* inference relation *δ for S is a partial function mapping every n-clause $\mathcal{C} \in S$ to a set of n-clauses in S such that $\mathcal{C}$ is deducible from $\delta(\mathcal{C})$ by one of the inference rules (in exactly one step).*

In practice this relation will be obtained from the inferences previously performed by the prover ($\mathcal{D} \in \delta(\mathcal{C})$ iff $\mathcal{D}$ is a parent of $\mathcal{C}$). The n-clauses $\mathcal{C}$ such that $\delta(\mathcal{C})$ is not defined are hypotheses, i.e., n-clauses occurring in the initial set. An inference relation induces a entailment relation $\vdash_\delta$ between subsets of S. Informally, we write $S' \vdash_\delta S''$ if all the n-clauses in S'' can be derived from n-clauses in $S' \cup S[\top]$ using inferences in δ. This is formalized as follows:

Definition 8. *Let S be a set of n-clauses and let δ be an inference relation for S. The relation $\vdash_\delta$ is the smallest relation between subsets of S such that $S' \vdash_\delta S''$ if for every n-clause $\mathcal{C} \in S''$, one of the following conditions holds:*

1. *$\mathcal{C} \in S'$.*
2. *$\delta(\mathcal{C})$ is defined and $S' \vdash_\delta \delta(\mathcal{C})$.*
3. *The constraint part of $\mathcal{C}$ is* `true`*.*

The intuition behind Condition 3 is that the n-clauses whose constraints are `true` are universal properties, which are valid independently of the value of n. Thus we assume that such n-clauses (once they have been proven, i.e., if they occur in S) can be used as hypotheses in any derivation.

Proposition 4. *Let S be a set of n-clauses and let δ be an inference relation for S. If $S' \vdash_\delta S''$ then $S' \cup S[\top] \models S'' \cup S[\top]$.*

In order to test entailment between clause sets, we introduce a notion of immediate entailment:

Definition 9. *An* immediate entailment relation *is a decidable relation $\sqsupseteq$ between n-clauses such that $\mathcal{C} \sqsupseteq \mathcal{D} \Rightarrow \mathcal{C} \models \mathcal{D}$. The relation $\sqsupseteq$ is extended to sets of n-clauses as follows: $S \sqsupseteq S'$ iff $\forall \mathcal{C}' \in S' \, \exists \mathcal{C} \in S, \mathcal{C} \sqsupseteq \mathcal{C}'$.*

In practice, $\sqsupseteq$ can be for instance the identity ($\mathcal{C} = \mathcal{D}$ up to a renaming of variables), the inclusion ($\mathcal{C} \subseteq \mathcal{D}$), or the subsumption relation $\leq_{sub}$.

We are now in position to define the notion of cycle, which is the syntactic pendant of the notion of inductive loop of Definition 6. It is defined relatively to the two relations $\sqsupseteq$ and δ.

Definition 10. *Let S be a set of clauses, let $\sqsupseteq$ be an immediate consequence relation and let δ be an inference relation on S. A pair of natural numbers (i,j) (with $j \neq 0$) is a* cycle *for S w.r.t. $\sqsupseteq$ and δ if there exist $S_{init}, S_{loop} \subseteq S$ such that $S_{init} \subseteq S[i], S_{loop} \subseteq S[i+j]$, $S_{init} \vdash_\delta \{n \not\simeq k \mid k \in [i, i+j-1]\}$, $S_{init} \vdash_\delta S_{loop}$ and $S_{loop} \sqsupseteq S_{init}\!\downarrow_j$.*

Example 2. Consider the derivation of Example 1. Assume that $\sqsupseteq$ is the identity relation. The pair $(0,1)$ is a cycle, with $S_{init} = \{5\}$ and $S_{loop} = \{7\}$. Indeed, the clause $n \not\simeq 0$ is derivable from Clause 5 and clauses not containing n, thus $S_{init} \vdash_\delta \{n \not\simeq 0\}$. Similarly, Clause 7 can be derived from Clause 5, together with clauses not containing n. Finally, we have $[p(x,y) \not\simeq \texttt{true} \mid n \simeq x]\!\downarrow_1 = [p(x,y) \not\simeq \texttt{true} \mid n \simeq \mathit{succ}(x)]$, hence $S_{loop} \sqsupseteq S_{init}\!\downarrow_1$ (assuming that $\sqsupseteq$ contains the identity relation). Similarly, $(0,2)$ and $(1,2)$ are also cycles, corresponding to the sets $\{5\}$, $\{9\}$ and $\{7\}$, $\{9\}$ respectively.

Theorem 2. *All cycles are inductive loops.*

Proof. Let (i,j) be a cycle for S, w.r.t. two relations $\sqsupseteq$ and δ. Let S_{init} and S_{loop} be the corresponding subsets of S. Let $S' = S_{init} \cup S[\top]$. We have $S_{init} \vdash_\delta \{n \not\simeq k\}$, for every $k \in [i, i+j-1]$, thus by Proposition 4, $S' \models n \not\simeq k$ (for every $k \in [i, i+j-1]$). Similarly, we have $S' \models S_{loop}$, hence $S' \models S_{init}\!\downarrow_j$ (since $\sqsupseteq$ is included in $\models$). But $S'\!\downarrow_j = S_{init}\!\downarrow_j \cup S[\top]\!\downarrow_j = S_{init}\!\downarrow_j \cup S[\top]$ (since the n-clauses whose constraint is $\texttt{true}$ are not affected by the $S\!\downarrow_j$ operation). Therefore $S' \models S'\!\downarrow_j$ and (i,j) is an inductive loop.

Theorems 1 and 2 entail the soundness of the following inference rule:

$$\text{Cycle elimination rule:} \qquad \frac{S}{n < i}$$

If (i,j) is a cycle for S w.r.t. some immediate consequence relation $\sqsupseteq$, and the inference relation δ induced by the previous inferences performed on S.

Theorem 2 gives a syntactic criterion to check whether a couple of fixed natural numbers defines a loop. We now show how to test efficiently that (i,j) is

Algorithm 1. CYCLE$_1(S, i, j)$

```
$S_0 \leftarrow \{n \not\simeq k, k \in [i, i+j[\}$
if $S[i] \not\vdash_\delta S_0$ then
    return false
end if
Choose a minimal set $S_{init} \subseteq S[i]$ s.t. $S_{init} \vdash_\delta S_0$
$S_{loop} \leftarrow \emptyset$
while $\exists \mathcal{C} \in S_{init} \mid S_{loop} \not\sqsupseteq \{\mathcal{C}{\downarrow_j}\}$ do
    if $\exists \mathcal{D} \in S[i+j] \mid \mathcal{D} \sqsupseteq \mathcal{C}{\downarrow_j}$ then
        Choose $\mathcal{D} \in S[i+j]$ such that $\mathcal{D} \sqsupseteq \mathcal{C}{\downarrow_j}$
        $S_{loop} \leftarrow S_{loop} \cup \{\mathcal{D}\}$
        if $S[i] \not\vdash_\delta \mathcal{D}$ then
            return false
        else
            Choose a set $S' \in S[i]$ such that $S' \vdash_\delta \{\mathcal{D}\}$
            $S_{init} \leftarrow S_{init} \cup S'$
        end if
    else
        return false
    end if
end while
return true
```

a cycle. Two distinct algorithms are presented. The algorithm CYCLE$_1$ is the most straightforward and uses a smallest fixpoint algorithm: it starts by considering the minimal possible set S_{init}, namely the set of n-clauses in $S[i]$ that are necessary to derive all the clauses $n \not\simeq k$ for $k \in [i, i+j[$. According to Definition 10, the condition $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$ must hold. Thus, for each clause $\mathcal{C}$ in S_{init}, the algorithm checks whether there exists a n-clause $\mathcal{D}$ in $S[i+j]$ such that $\mathcal{D} \sqsupseteq \mathcal{C}{\downarrow_j}$. If this is not the case, then (i, j) cannot be a cycle and the algorithm stops. Otherwise, all the ancestors of $\mathcal{D}$ occurring in $S[i]$ must be added to S_{init}, so that the condition $S_{init} \vdash_\delta S_{loop}$ (in Definition 10) holds. The algorithm runs until a fixpoint is reached, in which case a pair of sets of n-clauses (S_{init}, S_{loop}) satisfying the conditions of Definition 10 has been obtained. The drawback of this algorithm is that it is non-deterministic. Indeed, for a given n-clause $\mathcal{C}$, there may exist several n-clauses $\mathcal{D}$ satisfying the desired condition (unless $\sqsupseteq$ is the identity relation). Similarly, the ancestors of $\mathcal{D}$ in $S[i]$ are not unique and can be chosen arbitrarily. To ensure completeness all these branches must be explored, yielding an exponential number of immediate entailment tests (although the number of iteration steps is polynomial w.r.t. the size of S).

The algorithm CYCLE$_2$ is slightly more complex, and is based on a greatest fixpoint algorithm. The idea is to start by adding to S_{init} all the n-clauses in $S[i]$. Then S_{loop} is obtained by considering all the n-clauses in $S[i+j]$ that are generated from S_{init}. In order to ensure that the condition $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$ of Definition 10 holds, we remove from S_{init} the n-clauses $\mathcal{C}$ such that there is no n-clause $\mathcal{D}$ in $S[i+j]$ with $\mathcal{D} \sqsupseteq \mathcal{C}{\downarrow_j}$. If the removed n-clause $\mathcal{C}$ is an ancestor of a

Algorithm 2. $\textsc{Cycle}_2(S, i, j)$

```
$S_0 \leftarrow \{n \not\simeq k, k \in [i, i+j[\}$
$S_{init} \leftarrow S[i]$
if $S_{init} \not\vdash_\delta S_0$ then
    return false
end if
$S_{loop} \leftarrow \{\mathcal{D} \in S[i+j] \mid S_{init} \vdash_\delta \{\mathcal{D}\}\}$
while $\exists \mathcal{C} \in S_{init} \mid S_{loop} \not\sqsupseteq \{\mathcal{C}\downarrow_j\}$ do
    $S_{init} \leftarrow S_{init} \setminus \{\mathcal{C}\}$
    if $S_{init} \not\vdash_\delta S_0$ then
        return false
    end if
    Remove from $S_{loop}$ all the $n$-clauses $\mathcal{D}$ s.t. $S_{init} \not\vdash_\delta \{\mathcal{D}\}$
end while
return true
```

clause $n \not\simeq k$ for some $i \in [i, i+j[$ then no cycle possibly exists, and the algorithm stops. Otherwise, the deletion of $\mathcal{C}$ from S_{init} yields the removal of the n-clauses in S_{loop} that are generated from this clause (so that the invariant $S_{init} \vdash_\delta S_{loop}$ holds). This algorithm is deterministic and thus involves a polynomial number of immediate entailment tests (since it is clear that the number of iterations is polynomially bounded by the size of the initial set S). Its actual complexity depends of course on the relation $\sqsupseteq$: it is polynomial if $\sqsupseteq$ can be tested in polynomial time (for instance if $\sqsupseteq$ is the identity or inclusion relations). If $\sqsupseteq$ is the subsumption relation then it is exponential. The main drawback of $\textsc{Cycle}_2$ w.r.t. the previous algorithm $\textsc{Cycle}_1$ is that the handled clause sets are usually larger since the whole set of n-clauses must be considered right from the beginning. Thus the first algorithm may be more adapted to huge clause sets, or if the immediate entailment relation is the identity.

Theorem 3. *The algorithm* $\textsc{Cycle}_1$ *and* $\textsc{Cycle}_2$ *are terminating, sound and complete, i.e.,* $\textsc{Cycle}_1(S, i, j) = \texttt{true}$ *iff* $\textsc{Cycle}_2(S, i, j) = \texttt{true}$ *iff* (i, j) *is cycle for* S *(w.r.t. the relations* $\sqsupseteq$ *and* δ*)*

Proof. We consider the two algorithms separately.

Algorithm $\textsc{Cycle}_1$**:**

Termination is immediate since at each iteration step, the size of S_{init} increases strictly (and it is bounded by the size of the whole set of n-clauses). If $S[i] \not\models \{n \not\simeq k, k \in [i, i+j[\}$ then by Definition 10, (i, j) cannot be a cycle. Otherwise, according again to Definition 10, the set S_{init} must contain a set of n-clauses entailing $\{n \not\simeq k, k \in [i, i+j[\}$ (w.r.t. $\vdash_\delta$). The end-condition of the while loop ensures that $S_{loop} \sqsupseteq S_{init}\downarrow_j$. Furthermore, the invariant $S_{init} \vdash_\delta S_{loop}$ necessarily holds, since each time a clause $\mathcal{D}$ is added into S_{loop}, a set S' such that $S' \vdash_\delta \{\mathcal{D}\}$ is added to S_{init}. Finally, all the n-clauses that are added in S_{init} during the loop are in $S[i]$, thus the invariant $S_{init} \subseteq S[i]$ holds. Consequently, after the while loop, all the conditions of Definition 10 hold, and thus (i, j) must be a cycle. Conversely, if (i,j) is a cycle, then it is easy to check that a run of Algorithm $\textsc{Cycle}_1$ exists in which `false` is never returned. It suffices to choose the clauses $\mathcal{D}$ and S' in such a way that $\mathcal{D} \in S_{loop}$ and $S' \subseteq S_{init}$ (where S_{loop} and S_{init} correspond to the sets in Definition 10). This is always possible, since by definition we must have $S_{init} \vdash_\delta S_{loop}$ and $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$, with $S_{init} \subseteq S[i]$ and $S_{loop} \subseteq S[i+j]$.

Algorithm $\textsc{Cycle}_2$**:**

Termination is immediate since at each iteration step, the size of S_{init} decreases strictly. Again, if $S[i] \not\models \{n \not\simeq k, k \in [i, i+j[\}$ then by Definition 10, (i,j) cannot be a cycle. Otherwise, we must have $S_{init} \subseteq S[i]$ and $S_{loop} \subseteq S[i+j]$. As for the previous algorithm, the end-condition of the while loop ensures that $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$. Furthermore, the invariant $S_{init} \vdash_\delta S_{loop}$ still holds, by definition of the last instruction in the while-loop. Consequently, after the while loop, all the conditions of Definition 10 hold, and thus (i,j) must be cycle.

Conversely, if (i,j) is a cycle, then the algorithm returns `true`. Indeed, the sets S'_{init} and S'_{loop} corresponding to Definition 10 necessarily occur at each iteration step in the actual sets S_{init} and S_{loop} computed by the algorithm. By definition, no n-clause $\mathcal{C} \in S'_{init}$ can be removed from S_{init}, since we have $S_{loop} \sqsupseteq \{\mathcal{C}{\downarrow_j}\}$. Similarly, no clause $\mathcal{D} \in S'_{loop}$ can be deleted from S_{loop} since we must have $S_{init} \vdash_\delta \{\mathcal{D}\}$. Therefore, the condition $S_{init} \not\vdash_\delta S_0$ can never hold (since $S'_{init} \vdash_\delta S_0$).

Example 3. We consider the following clause set:

$$\begin{array}{rc}
1 & p(x) \not\simeq \mathtt{true} \vee p(succ(x)) \simeq \mathtt{true} \\
2 & q(x) \simeq \mathtt{true} \vee p(succ(x)) \simeq \mathtt{true} \\
3 & f(succ(x)) \simeq f(x) \\
4 & p(0) \simeq \mathtt{true} \\
5 & [p(x) \not\simeq \mathtt{true} \mid n \simeq x] \\
6 & [f(x) \simeq a \mid n \simeq x] \\
7 & [g(x) \simeq a \mid n \simeq x]
\end{array}$$

The following clauses are derived:

$$\begin{array}{rcr}
8 & [\Box \mid n \simeq 0] & \text{(superposition, 4, 5)} \\
9 & [p(x) \not\simeq \mathtt{true} \mid n \simeq succ(x)] & \text{(superposition, 1,5)} \\
10 & [f(x) \simeq a \mid n \simeq succ(x)] & \text{(superposition, 3,6)} \\
11 & [q(x) \simeq \mathtt{true} \mid n \simeq succ(x)] & \text{(superposition, 2,5)}
\end{array}$$

We illustrate how the two algorithms run on this example. Note that Clauses $1,4,5$ are sufficient for unsatisfiability (Clause 5 asserts that $\neg p(n)$ holds for some natural number n, which is impossible since Clauses 1 and 4 entail that $p(succ^k(0))$ holds for every $k \in \mathbb{N}$), the other clauses are added only to illustrate how the algorithms work and to emphasize the differences between them. We take $i = 0$, $j = 1$ and we use the identity as the immediate consequence relation $\sqsupseteq$. We have $S_0 = \{8\}$, $S[i] = \{5,6,7\}$, $S[i+j] = \{9,10,11\}$, $S[\bot] = \{1,2,3,4\}$.

Algorithm $\textsc{Cycle}_1$**:** The algorithm first chooses a set of clauses $S_{init} \subseteq S[i]$ such that $S_{init} \vdash_\delta S_0$. The parents of Clause 8 are $4 \in S[\bot]$ and $5 \in S[i]$, thus according to Definition 8 we have $\{5\} \vdash_\delta \{8\}$. Therefore, S_{init} is set to $\{5\}$. Then S_{loop} is initialized to $\emptyset$. Clause 5 occurs in S_{init} and we have $5{\downarrow_1} \not\in S_{loop}$ (since $S_{loop} = \emptyset$). Thus the algorithm chooses a clause $\mathcal{D} \in S[i+j]$ such that $\mathcal{D} \sqsupseteq 5{\downarrow_1}$. Clause 5 is $[p(x) \not\simeq \mathtt{true} \mid n \simeq x]$, thus $5{\downarrow_1}$ is $[p(x) \not\simeq \mathtt{true} \mid n \simeq succ(x)]$, hence $5{\downarrow_1} = 9$. Therefore, the only solution is $\mathcal{D} = 9$. This clause is added to S_{loop}. The algorithm checks that $S[i] \vdash_\delta \mathcal{D}$ and adds to S_{init} a minimal set of clauses S' such that $S' \vdash_\delta \{\mathcal{D}\}$. Clause 9 is deduced from Clause 1, which occurs in $S[\bot]$, and Clause 5, which occurs in $S[i]$, thus the only solution is $S' = \{5\}$. Therefore S_{init} is not affected (since it already contains 5). The while-loop ends because the only clause in $S_{init}{\downarrow_1}$ occurs in S_{loop}. The pair $(0,1)$ is a cycle, corresponding to the sets $S_{init} = \{5\}$ and $S_{loop} = \{9\}$.

Algorithm Cycle$_2$**:** S_{init} is initialized with $S[i]$, i.e. $\{5,6,7\}$. Then the algorithm checks that $S_{init} \vdash_\delta S_0$, and initializes S_{loop} with the set of clauses $\mathcal{D} \in S[i+j]$ such that $S_{init} \vdash_\delta \mathcal{D}$. All the clauses in $S[i+j]$ are obtained from clauses in S_{init} and $S[\bot]$, thus we have $S_{loop} = \{9,10,11\}$. Then the algorithm checks whether S_{init} contains a clause $\mathcal{C}$ such that $\mathcal{C}{\downarrow_j} \not\in S_{loop}$. The only clause satisfying this condition is Clause 7. Thus this clause is removed from S_{init}, yielding $\{5,6\}$ and the clauses $\mathcal{D}$ such that $\{5,6\} \not\vdash_\delta \mathcal{D}$ are removed from S_{loop}. Here all the clauses in S_{loop} can be deduced from $\{5,6\}$ and clauses in $S[\bot]$ thus S_{loop} is not affected. Then the algorithm stops and returns `true`. The obtained sets are $S_{init} = \{5,6\}$ and $S_{loop} = \{9,10\}$. Note that, compared to the previous case, the sets contain an additional clause (namely 6/10), which occurs in the generated inductive invariant, but actually plays no role in the proof (these clauses can be identified and eliminated afterwards by applying reachability analysis algorithms on the inference graph).

6 Implementation

Our calculus has been implemented as a research prototype, using the system Prover9 [17] as an inference engine. While any other superposition-based prover could be used instead, the system is not used as a mere "black box": the procedures and data-structures had to be adapted in order to handle the specific features of the calculus: arithmetic constraints, normalization of clauses, etc. The program uses the usual "given clause algorithm" of Prover9, and calls Algorithm Cycle$_2$ to check whether a given pair (i,j) is a cycle, using the subsumption relation as an immediate consequence relation. The test is triggered at each iteration of the main loop, and only if all the clauses $n \not\simeq k$ for $k \in [0, i+j[$ have been generated (thus a cycle is detected only if this leads to an immediate stop). A refutation is obtained if the system generates a set of n-clauses of the form $\{\Box\}$ or $\{n \not\simeq 0, \dots, n \not\simeq k-1, n < k\}$, which is obviously unsatisfiable. The last clause $n < k$ is usually derived by cycle detection (with $k = i+j$), but it can also be derived by the superposition calculus alone, in simple cases in which the theorem can be proven without induction. If the system is fed with an unsatisfiable set of standard clauses then the empty clause $\Box$ can be generated as usual. Our proof procedure is not complete in general (the logic is not semi-decidable).

We use heuristics to preserve the partial completeness results in [16]. For instance a greater weight is associated with the symbol *succ* to ensure that the literals containing a maximal arithmetic expression are selected with the highest priority, and some inferences are blocked to ensure that $S[i] \models S[i+1]$.

Now, let's prove the theorem:

$$\forall n \in \mathbb{N}\ \forall a_1, \dots, a_n \quad a_1 \times a_2 \times \cdots \times a_n = a_n \times a_{n-1} \times \cdots \times a_1 \tag{1}$$

We show the corresponding input file:

```
formulas(sos).
N(x) | p(x) != q(x).
p(0)=1.
q(0)=1.
p(s(x)) = p(x)*a(x).
q(s(x)) = a(x)*q(x).
*(x,1)=x.
*(1,x)=x.
x*y= u*v | x!=u | y != v.
x*y = y*x.
end_of_list.
```

Our tool has almost the same input format than Prover9, we just have to add the constraints to the clauses, a constraint of the form $n \not\simeq t$ (where t is a term of sort `nat`) is written $N(t)$ and attached to the clause as a literal. The first clause of the input file corresponds to $[p(x) \not\simeq q(x) \mid n \simeq x]$, which is also the negation of (1), p(x) and q(x) encode the terms $a_1 \times \ldots \times a_n$ and $a_n \times \ldots \times a_1$ respectively. We show the output file generated by our tool:

```
============================== PROOF =================================
% Proof at 0.02 seconds.
% Given clauses 17.
S_init  :
(67: N(v0) | -=(0,1) | -=(1,v0) .
   2: N(v0) | -=(q(v0),p(v0)).)
S_loop :
(107: N(s(v0)) | -=(0,1) | -=(1,v0) .
   85: N(s(v0)) | -=(q(v0),p(v0)).)
The empty clauses  :
 (12: N(0).)
============================== end of proof ==========================
```

The output file contains the running time, the number of given clauses, the two clause sets S_{init}, S_{loop} and finally the pure constraint clause $N(0)$ which corresponds to the clause $[\square \mid n \simeq 0]$. As in Example 3, the obtained inductive invariant contains an additional clause that plays no role in the derivation.

7 Experimentation

In this section we provide some examples of application of our work. All the presented problems require induction and thus are out of the scope of first-order theorem provers. We first present some examples in propositional logic. We consider an n-bit sequential adder circuit i.e. a circuit which computes the sum of two bit-vectors of length n. Such a circuit is built by composing n 1-bit adders. The i^{th} bits of each operand are written p_i and q_i. r_i is the i^{th} bit of the result and c_{i+1} is carried over to the next bit (thus $c_1 = 0$). We set the notations ($\oplus$ denotes exclusive or): $\mathit{Sum}_i(p,q,c,r) \stackrel{\text{def}}{=} r_i \Leftrightarrow (p_i \oplus q_i) \oplus c_i$ and $\mathit{Carry}_i(p,q,c) \stackrel{\text{def}}{=} c_{i+1} \Leftrightarrow (p_i \wedge q_i) \vee (c_i \wedge p_i) \vee (c_i \wedge q_i)$. Then the formula: $\mathit{Adder}(p,q,c,r) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} \mathit{Sum}_i(p,q,c,r) \wedge \bigwedge_{i=1}^{n} \mathit{Carry}_i(p,q,c) \wedge \neg c_1$ with the constraint $n \geq 1$, schematises the adder circuit (it states that r encodes the sum of p and q). In order to test the satisfiability of such schemata of propositional formulæ, we have implemented an algorithm transforming automatically (in polynomial time) any propositional schema built on iterated connectives of the form $\bigwedge_{i=a}^{n+b} \phi$ or $\bigvee_{i=a}^{n+b} \phi$ (such as the ones modeling the Adder circuit) into a sat-equivalent set of n-clauses. This algorithm works by introducing a monadic predicate (of domain `nat`) for every iteration occurring in the initial formula, and by adding axioms to specify the interpretation of these predicates by induction on the natural numbers. For instance, the schema $\bigvee_{i=0}^{n} p_i$ can be denoted by the atom $[q(x) \simeq \mathtt{true} \mid n \simeq x]$, with the axioms: $\{q(0) \not\simeq \mathtt{true} \vee p(0) \simeq \mathtt{true}, q(\mathit{succ}(x)) \not\simeq \mathtt{true} \vee p(\mathit{succ}(x)) \simeq \mathtt{true} \vee q(x) \simeq \mathtt{true}\}$ (the formal description of the transformation algorithm is omitted due to space restrictions, it can be found in [2]). Several properties of the *Adder* can then be automatically checked, such as commutativity or associativity. We have encoded two different versions of the Adder (the carry propagate and ripple-carry adders respectively) and proved some elementary properties of these circuits.

We have also considered examples coming from an interesting application of schemata languages in proof theory, developed in the context of the ASAP project (see `http://membres-lig.imag.fr/peltier/ASAP/`). The method CERES (see for instance [3]) is an algorithm for cut-elimination in first-order logic that is more efficient that the standard (reductive) approach. It works by extracting from the considered (non-analytic) proof π an unsatisfiable set of clauses $S(\pi)$, called the *characteristic set* of π, which is defined in such a way that any resolution proof of $S(\pi)$ can be automatically transformed into an analytic proof of S. It has been extended to schemata of first-order proofs in [9,19,10], in order to handle mathematical proofs using induction (which cannot be expressed in first-order logic and which, as well-known, do not admit cut elimination algorithms). The obtained characteristic set is then not a set of clauses in the usual sense, but rather a schema of clause sets, which can be expressed as a set of n-clauses, and handled using our calculus. We provide the running times for the characteristic sets obtained from simple proofs (the formal definition of the schemata is omitted for conciseness, the purely propositional ones can be found in the RegSTAB webpage at `http://regstab.forge.ocamlcore.org/` and the first-order one can be found in [19,10]). Finally we consider some simple inductive properties, for instance we prove that if we perform an arbitrary number of permutations on a sequence containing an element a then the final sequence still contains a.

The obtained results are depicted below. We provide for each example, the running time, the number of calls to Cycle$_2$ and the number of generated clauses.

Example	Time (s)	# of calls to Cycle$_2$	# generated clauses
Ripple-carry adder ($A + 0 = A$)	0.48	336	33833
Ripple-carry adder (commutativity)	0.03	102	2003
Ripple-carry adder (associativity)	0.09	207	10154
Ripple-carry adder ($3 + 4 = 7$)	0.06	71	9989
Unicity of the result (ripple-carry)	0.7	150	50901
Carry-propagate adder (commutativity)	0.02	14	1980
Carry-propagate adder (associativity)	0.01	20	3972
Equivalence between the ripple-carry and the carry-propagate adders	0.03	14	1980
CERES ex1 (Propositional)	0.01	40	995
CERES ex2 (Propositional)	0.03	216	4106
CERES (First order)	0.01	23	49
Totality of $<$ ($n_1 \geq n_2 \vee n_1 < n_2$)	0.01	47	185
$\bigwedge_{i=1}^{n} p_i > 0 \Rightarrow p_1 \times \cdots \times p_n > 0$	0.01	8	59
Permutation (triplet)	0.01	17	280

The results show that the cycle detection algorithm is efficient, even for sets containing thousands of clauses.

8 Conclusion

We have presented a method to enrich superposition-based theorem proving with inductive reasoning capabilities. To this purpose, we have devised algorithms to detect cycles in the superposition derivation in an automatic way. These cycles correspond to inductive invariants and allow one to prune infinite superposition derivations. Our method has been implemented and some examples of application have been presented. Future work includes the extension of the implementation, for instance by devising refined criteria for triggering the application of the cycle detection procedure or by introducing new techniques for performing this detection in an incremental way.

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. Aravantinos, V., Echenim, M., Peltier, N.: A resolution calculus for first-order schemata. Fundamenta Informaticae (accepted for publication, to appear, 2013)
3. Baaz, M., Leitsch, A.: Towards a clausal analysis of cut-elimination. J. Symb. Comput. 41(3-4), 381–410 (2006)
4. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational theorem proving for hierachic first-order theories. Appl. Algebra Eng. Commun. Comput. 5, 193–212 (1994)
5. Barthe, G., Stratulat, S.: Validation of the javacard platform with implicit induction techniques. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 337–351. Springer, Heidelberg (2003)
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. 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)
8. Comon, H.: Inductionless induction. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 14, pp. 913–962. North-Holland (2001)
9. Dunchev, T.: Automation of cut-elimination in proof schemata. PhD thesis, T.U. Vienna (2012)
10. Dunchev, T., Leitsch, A., Rukhaia, M., Weller, D.: Ceres for first-order schemata, Research Report (2013), `http://arxiv.org/abs/1303.4257`
11. Falke, S., Kapur, D.: Rewriting induction + linear arithmetic = decision procedure. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 241–255. Springer, Heidelberg (2012)
12. 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)
13. Giesl, J., Kapur, D.: Decidable classes of inductive theorems. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 469–484. Springer, Heidelberg (2001)
14. Giesl, J., Kapur, D.: Deciding inductive validity of equations. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 17–31. Springer, Heidelberg (2003)
15. Horbach, M., Weidenbach, C.: Superposition for fixed domains. ACM Trans. Comput. Logic 11(4), 1–35 (2010)
16. Kersani, A., Peltier, N.: Completeness and Decidability Results for First-Order Clauses with Indices. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 58–75. Springer, Heidelberg (2013)
17. McCune, W.: Prover9 and mace4 (2005–2010), `http://www.cs.unm.edu/~mccune/prover9/`
18. Robinson, A., Voronkov, A. (eds.): Handbook of Automated Reasoning. North-Holland (2001)
19. Rukhaia, M.: CERES in Proof Schemata. PhD thesis, T.U. Vienna (2012)
20. Stratulat, S.: Automatic 'Descente infinie' induction reasoning. In: Beckert, B. (ed.) TABLEAUX 2005. LNCS (LNAI), vol. 3702, pp. 262–276. Springer, Heidelberg (2005)

Definability of Accelerated Relations in a Theory of Arrays and Its Applications

Francesco Alberti[1], Silvio Ghilardi[2], and Natasha Sharygina[1]

[1] Formal Verification Lab, University of Lugano, Lugano, Switzerland
[2] Università degli Studi di Milano, Milan, Italy

Abstract. For some classes of guarded ground assignments for arrays, we show that accelerations (i.e. transitive closures) are definable in the theory of arrays via $\exists^*\forall^*$-first order formulae. We apply this result to model checking of unbounded array programs, where the computation of such accelerations can be used to prevent divergence of reachability analysis. To cope with nested quantifiers introduced by acceleration preprocessing, we use simple instantiation and refinement strategies during backward search analysis. Our new acceleration technique and abstraction/refinement loops are mutually beneficial: experiments conducted with the SMT-based model checker MCMT attest the effectiveness of our approach where acceleration and abstraction/refinement technologies fail if applied alone.

1 Introduction

Transitive closure is a logical construct that is far beyond first order logic: either infinite disjunctions or higher order quantifiers or, at least, fixpoints operators are required to express it. Indeed, due to the compactness of first order logic, transitive closure (even modulo the axioms of a first order theory) is first-order definable only in trivial cases. These general results do not hold if we define a theory as a class of structures $\mathcal{C}$ over a given signature[1]. Such definition is different from the "classical" one where a theory is identified as a set of axioms. By taking a theory as a class of structures the property of compactness breaks, and it might well happen that transitive closure becomes first-order definable (the first order definition being valid just inside the class $\mathcal{C}$ - which is often reduced to a single structure).

In this paper we consider the extension of Presburger arithmetic with free unary function symbols. Inside Presburger arithmetic, various classes of relations are known to have definable *acceleration*[2] (see related work section below). In our combined setting, the presence of free function symbols introduces a novel feature that, for instance, limits decidability to controlled extensions of the quantifier-free fragment [16,23]. In this paper we show that in such theory some classes of relations admit a definable acceleration.

[1] Such definition is widely adopted in the SMT literature [8].

[2] 'acceleration' is the name usually adopted in the formal methods literature to denote transitive closure.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 23–39, 2013.

The theoretical problem of studying the definability of accelerated relations has an important application in program verification. The theory we focus on is widely adopted to represent programs handling arrays, where free functions model arrays of integers. In this application domain, the accelerated counterpart of relations encoding systems evolution (e.g., loops in programs) allows to compute 'in one shot' the reachable set of states after an arbitrary but finite number of execution steps. This has the great advantage of keeping under control sources of (possible) divergence arising in the reachability analysis.

The contributions of the paper are many-fold. First, we show that inside the combined theory of Presburger arithmetic augmented with free function symbols, the acceleration of some classes of relations – corresponding, in our application domain, to relations involving arrays and counters – can be expressed in first order language. This result comes at a price of allowing nested quantifiers. Such nested quantification can be problematic in practical applications. To address this complication, as a second contribution of the paper, we show how to take care of the quantifiers added by the accelerating procedure: the idea is to import in this setting the so-called *monotonic abstraction* technique [1, 2]. Such technique has been reinterpreted and analyzed in a declarative context in [5]: from a logical point of view, it amounts to a restricted form of *instantiation for universal quantifiers*. Third, we show that the ability to compute accelerated relations is greatly beneficial in program verification. In particular, one of the biggest problems in verifying safety properties of array programs is designing procedures for the synthesis of relevant quantified predicates. In typical sequential programs (like those illustrated in Fig.1), the guarded assignments used to model the program instructions are ground and, as a consequence, the formulae representing backward reachable states are ground too. However, the invariants required to certify the safety of such programs contain quantifiers. Our acceleration procedure is able to supply the required quantified predicates. Our experimentation attests that abstraction/refinement-based strategies widely used in verification benefit from accelerated transitions. In programs with nested loops, as the `allDiff` procedure of Fig.1 for example, the ability to accelerate the inner loop simplifies the structure of the problem, allowing abstraction to converge during verification of the entire program. For such programs, abstraction/refinement or acceleration approaches taken in isolation are not sufficient; reachability analysis converges only if they are combined together.

Related Work. To the best of our knowledge, the only work addressing the problem of accelerating relations involving arrays is [13]. The approach used in this paper seems to be unable to handle properties of common interest with more than one quantified variable (e.g., "sortedness") and is limited to programs without nested loops. Our technique is not affected by such limitations and can successfully handle examples outside the scope of [13].

Inside Presburger arithmetic, various classes of relations are known to have definable acceleration: these include relations that can be formalized as difference bounds constraints [15, 20], octagons [12] and finite monoid affine transformations [21] (The paper [14] presents a general approach covering all these

```
function allDiff ( int a[N] ) :
  r = true;
  for (i = 1; $i < N \wedge r$; i++)
    for (j = i-1; $j \geq 0 \wedge r$; j--)
      if (a[i] = a[j]) r = false;
  assert $\left( r \rightarrow \left( \forall x, y (0 \leq x < y < N) \rightarrow (a[x] \neq a[y]) \right) \right)$
```

(a)

```
function Reverse ( int I[N + 1]; int O[N + 1]; int c ) :
  c = 0;
  while ($c \neq N + 1$) {O[c] = I[$N - c$]; c++; }
  assert $\left( \forall x \geq 0, y \geq 0 \; (x + y = N \rightarrow I[x] = O[y]) \right)$
```

(b)

Fig. 1. Motivating examples

domains). Acceleration for relations over Presburger arithmetic has been also plugged into abstraction/refinement loop for verifying integer programs [17, 27].

We recall that acceleration has also been applied fruitfully in the analysis of real time systems (e.g., [9, 26]), to compactly represent the iterated execution of cyclic actions (e.g., polling-based systems) and address fragmentation problems.

Our work can be proficiently combined with SMT-based techniques for the verification of programs, as it helps avoiding the reachability analysis divergence when it comes to abstraction of programs with arrays of unknown length. Since the technique mostly operates at pre-processing level (we add to the system accelerated transitions by collapsing branches of loops handling arrays), *we believe that our technique is compatible with most approaches* proposed in array-based software model checking. We summarize some of these approaches below, without pretending of being exhaustive.

The vast majority of software model-checkers implement abstraction-refinement algorithms (e.g., [7,19,25]). *Lazy Abstraction with Interpolants* [31] is one of the most effective frameworks for unbounded reachability analysis of programs. It relies on the availability of interpolation procedures (nowadays efficiently embedded in SMT-Solvers [18]) to generate new predicates as (quantifier-free) interpolants for refining infeasible counterexamples.

For programs with arrays of unknown length the classical interpolation-based lazy abstraction works only if there is a support to handle quantified predicates [3] (the approach of [3] is the basis of our experiments below). Effectiveness and performances of abstraction/refinement approaches strongly depend on their ability in generating the "right" predicates to stop divergence of verification procedures. In case of programs with arrays, this quest can rely on *ghost variables* [22] retrieved from the post-conditions, on the backward propagation of post-conditions along spurious counterexamples [34] or can be constraint-based [10, 35]. Recently, constraint-based techniques have been significantly extended to the generation of loop invariants outside the array property fragment [30]. This solution exploits recent advances in SMT-Solving, namely those devoted to finding solutions of constraints over non-linear integer arithmetic [11]. Other ways to generate predicates are by means of *saturation-based* theorem provers [29, 32] or interpolation procedures [3, 28].

All the aforementioned techniques suffer from a certain degree of randomness due to the fact that detecting the "right" predicate is an undecidable problem. For example, predicate abstraction approaches (i.e., [3, 4, 34]) fail verifying the procedures in Fig.1, which are commonly considered to be challenging for verifiers because they cause divergence[3]. Acceleration, on the other side, provides a precise and systematic way for addressing the verification of programs. Its combination, as a preprocessing procedure, with standard abstraction-refinement techniques allows to successfully solve challenging problems like the ones in Fig.1.

The paper is structured as follows: Section 2 recalls the background notions about Presburger arithmetic and extensions. In order to identify the classes of relations whose acceleration we want to study, we are guided by software model checking applications. To this end, we provide in Section 3 a classification of the guarded assignments we are interested in. Section 4 demonstrates the practical application of the theoretical results. In particular, it presents a backward reachability procedure and shows how to incorporate acceleration with monotonic abstraction in it. The details of the theoretical results are presented later. The main definability result for accelerations is in Section 6, while Section 5 introduces the abstract notion of an iterator. Section 7 discusses our experiments and Section 8 concludes the paper.

2 Preliminaries

We work in Presburger arithmetic enriched with free function symbols and with definable function symbols (see below); when we speak about validity or satisfiability of a formula, we mean *satisfiability and validity in all structures having the standard structure of natural numbers as reduct.* Thus, satisfiability and validity are decidable if we limit to quantifier-free formulæ (by adapting Nelson-Oppen combination results [33, 36]), but may become undecidable otherwise (because of the presence of free function symbols).

We use $x, y, z, \dots$ or $i, j, k, \dots$ for variables; $t, u, \dots$ for terms, $c, d, \dots$ for free constants, $a, b, \dots$ for free function symbols, $\phi, \psi, \dots$ for *quantifier-free* formulæ. Bold letters are used for tuples and $|-|$ indicates tuples length; hence for instance $\mathbf{u}$ indicates a tuple of terms $u_1, \dots, u_m$, where $m = |\mathbf{u}|$. These tuples may contain repetitions. For variables, we use underlined letters $\underline{x}, \underline{y}, \dots, \underline{i}, \underline{j}, \dots$ to indicate tuples without repetitions. Vector notation can also be used for equalities: if $\mathbf{u} = u_1, \dots, u_n$ and $\mathbf{v} = v_1, \dots, v_n$, we may use $\mathbf{u} = \mathbf{v}$ to mean the formula $\bigwedge_{i=1}^{n} u_i = v_i$.

If we write $t(x_1, \dots, x_n), \mathbf{u}(x_1, \dots, x_n), \phi(x_1, \dots, x_n)$ (or $t(\underline{x}), \mathbf{u}(\underline{x}), \phi(\underline{x}), \dots$, in case $\underline{x} = x_1, \dots, x_n$), we mean that the term t, the tuple of terms $\mathbf{u}$, the quantifier-free formula ϕ contain variables only from the tuple $x_1, \dots, x_n$. Similarly, we may use $t(\mathbf{a}, \mathbf{c}, \underline{x}), \phi(\mathbf{a}, \mathbf{c}, \underline{x}), \dots$ to mean *both* that the term t or the

[3] The procedure `Reverse` outputs to the array `O` the reverse of the array `I`; the procedure `allDiff` checks whether the entries of the array `a` are all different. Many thanks to Madhusudan Parthasarath and his group for pointing us to challenging problems with arrays of unknown length, including the `allDiff` example.

quantifier-free formula ϕ have free variables included in $\underline{x}$ *and* that the free function, free constants symbols occurring in them are among $\mathbf{a}, \mathbf{c}$. Notations like $t(\mathbf{u}/\underline{x}), \phi(\mathbf{u}/\underline{x}), \dots$ or $t(u_1/x_1, \dots, u_n/x_n), \phi(u_1/x_1, \dots, u_n/x_n), \dots$ - or occasionally just $t(\mathbf{u}), \phi(\mathbf{u}), \dots$ if confusion does not arise - are used for simultaneous substitutions within terms and formulæ. For a given natural number n, we use the standard abbreviations $\bar{n}$ and $n * y$ to denote the numeral of n (i.e. the term $s^n(0)$, where s is the successor function) and the sum of n addends all equal to y, respectively. If confusion does not arise, we may write just n for $\bar{n}$.

By a *definable function symbol*, we mean the following. Take a quantifier-free formula $\phi(\underline{j}, y)$ such that $\forall \underline{j} \exists! y \phi(\underline{j}, y)$ is valid ($\exists! y$ stands for 'there is a unique y such that ...'). Then a definable function symbol F (defined by ϕ) is a fresh function symbol, matching the length of $\underline{j}$ as arity, which is constrained to be interpreted in such a way that the formula $\forall y. F(\underline{j}) = y \leftrightarrow \phi(\underline{j}, y)$ is true. The addition of definable function symbols does not affect decidability of quantifier-free formulæ and can be used for various purposes, for instance in order to express directly case-defined functions, array updates, etc. For instance, if a is a unary free function symbol, the term $wr(a, i, x)$ (expressing the update of the array a at position i by over-writing x) is a definable function; formally, we have $\underline{j} := i, x, j$ and $\phi(\underline{j}, y)$ is given by $(j = i \wedge y = x) \vee (j \neq i \wedge y = a(j))$. This formula $\phi(\underline{j}, y)$ (and similar ones) is usually written as

$$y \;=\; (\texttt{if } j = i \texttt{ then } x \texttt{ else } a(j))$$

to improve readability. Another useful definable function is integer division by a fixed natural number n: to show that integer division by n is definable, recall that in Presburger arithmetic we have that $\forall x \; \exists! y \; \bigvee_{r=0}^{n-1} (x = n * y + r)$ is valid.

3 Programs Representation

As a first step towards our main definability result, we provide a classification of the relations we are interested in. Such relations are guarded assignments required to model programs handling arrays of unknown length.

In our framework a *program* $\mathcal{P}$ is represented by a tuple $(\mathbf{v}, l_I, l_E, T)$. The tuple $\mathbf{v} := \mathbf{a}, \mathbf{c}, pc$ models system variables. Formally, we have that

- the tuple $\mathbf{a} = a_1, \dots, a_s$ contains free unary function symbols, i.e., the arrays manipulated by the program;
- the tuple $\mathbf{c} = c_1, \dots, c_t$ contains free constants, i.e., the integer data manipulated by the program;
- the additional free constant pc (called *program counter*) is constrained to range over a finite set $L = \{l_1, ..., l_n\}$ of *program locations* over which we distinguish the *initial* and *error* locations denoted by l_I and l_E, respectively.

T is a set of finitely many formulæ $\{\tau_1(\mathbf{v}, \mathbf{v}'), \dots, \tau_r(\mathbf{v}, \mathbf{v}')\}$ called *transition formulæ* representing the program's body (here $\mathbf{v}'$ are renamed copies of the variable tuple $\mathbf{v}$ representing the next-state variables). $\mathcal{P} = (\mathbf{v}, l_I, l_E, T)$ is *safe* iff there is no satisfiable formula like

$$(pc^0 = l_I) \wedge \tau_{i_1}(\mathbf{v}^0, \mathbf{v}^1) \wedge \cdots \wedge \tau_{i_N}(\mathbf{v}^{N-1}, \mathbf{v}^N) \wedge (pc^N = l_E)$$

where $\mathbf{v}^0, \ldots, \mathbf{v}^N$ are renamed copies of $\mathbf{v}$ and each τ_{i_h} belongs to T.

Sentences denoting sets of states reachable by $\mathcal{P}$ can be:

- *ground* sentences, i.e., sentences of the kind $\phi(\mathbf{a}, \mathbf{c}, pc)$;
- Σ_1^0*-sentences*, i.e., sentences of the form $\exists \underline{i}.\ \phi(\underline{i}, \mathbf{a}, \mathbf{c}, pc)$;
- Σ_2^0*-sentences*, i.e., sentences of the form $\exists \underline{i} \forall \underline{j}.\ \phi(\underline{i}, \underline{j}, \mathbf{a}, \mathbf{c}, pc)$.

We remark that in our context satisfiability can be fully decided only for ground sentences and Σ_1^0-sentences (by Skolemization, as a consequence of the general combination results [33,36]), while only subclasses of Σ_2^0-sentences enjoy a decision procedure [16,23]. Transition formulæ can also be classified in three groups:

- *ground assignments*, i.e., transitions of the form

$$pc = l \ \wedge\ \phi_L(\mathbf{c}, \mathbf{a}) \ \wedge\ pc' = l' \ \wedge\ \mathbf{a}' = \lambda j.\ G(\mathbf{c}, \mathbf{a}, j) \ \wedge\ \mathbf{c}' = H(\mathbf{c}, \mathbf{a}) \qquad (1)$$

- Σ_1^0*-assignments*, i.e., transitions of the form

$$\exists \underline{k} \begin{pmatrix} pc = l \ \wedge\ \phi_L(\mathbf{c}, \mathbf{a}, \underline{k}) \ \wedge\ pc' = l' \ \wedge \\ \mathbf{a}' = \lambda j.\ G(\mathbf{c}, \mathbf{a}, \underline{k}, j) \ \wedge\ \mathbf{c}' = H(\mathbf{c}, \mathbf{a}, \underline{k}) \end{pmatrix} \qquad (2)$$

- Σ_2^0*-assignments*, i.e., transitions of the form

$$\exists \underline{k} \begin{pmatrix} pc = l \ \wedge\ \phi_L(\mathbf{c}, \mathbf{a}, \underline{k}) \ \wedge\ \forall \underline{j}\ \psi_U(\mathbf{c}, \mathbf{a}, \underline{k}, \underline{j}) \ \wedge \\ pc' = l' \ \wedge\ \mathbf{a}' = \lambda j.\ G(\mathbf{c}, \mathbf{a}, \underline{k}, j) \wedge\ \mathbf{c}' = H(\mathbf{c}, \mathbf{a}, \underline{k}) \end{pmatrix} \qquad (3)$$

where $G = G_1, \ldots, G_s$, $H = H_1, \ldots, H_t$ are tuples of definable functions (vectors of equations like $\mathbf{a}' = \lambda j.\ G(\mathbf{c}, \mathbf{a}, \underline{kj})$ can be replaced by the corresponding first order sentences $\forall j.\ \bigwedge_{h=1}^{s} a_h'(j) = G_h(\mathbf{c}, \mathbf{a}, \underline{k}, j))$.

The *composition* $\tau_1 \circ \tau_2$ of two transitions $\tau_1(\mathbf{v}, \mathbf{v}')$ and $\tau_2(\mathbf{v}, \mathbf{v}')$ is expressed by the formula $\exists \mathbf{v}_1 (\tau_1(\mathbf{v}, \mathbf{v}_1) \wedge \tau_2(\mathbf{v}_1, \mathbf{v}'))$ (notice that composition may result in an inconsistent formula, e.g., in case of location mismatch). The *preimage* $Pre(\tau, K)$ of the set of states satisfying the formula $K(\mathbf{v})$ along the transition $\tau(\mathbf{v}, \mathbf{v}')$ is the set of states satisfying the formula $\exists \mathbf{v}'(\tau(\mathbf{v}, \mathbf{v}') \wedge K(\mathbf{v}'))$. The following proposition is proved by straightforward syntactic manipulations:

Proposition 1. *Let τ, τ_1, τ_2 be transition formulæ and let $K(\mathbf{v})$ be a formula. We have that:* (i) *if τ_1, τ_2, τ, K are ground, then $\tau_1 \circ \tau_2$ is a ground assignment and $Pre(\tau, K)$ is a ground formula;* (ii) *if τ_1, τ_2, τ, K are Σ_1^0, then $\tau_1 \circ \tau_2$ is a Σ_1^0-assignment and $Pre(\tau, K)$ is a Σ_1^0-sentence;* (iii) *if τ_1, τ_2, τ, K are Σ_2^0, then $\tau_1 \circ \tau_2$ is a Σ_2^0-assignment and $Pre(\tau, K)$ is a Σ_2^0-sentence.*

4 Backward Search and Acceleration

This section demonstrates the practical applicability of the theoretical results of the paper in program verification. In particular, it presents the application of the accelerated transitions during reachability analysis for guarded assignments representing programs handling arrays. For readability, we first present a basic reachability procedure. We subsequently analyze the divergence problems and show how acceleration can be applied to solve them. Acceleration application is not straightforward, though. The presence of accelerated transitions might generate undesirable Σ^0_2-sentences. The solution we propose is to over-approximate such sentences by adopting a selective instantiation schema, known in literature as *monotonic abstraction*. An enhanced reachability procedure integrating acceleration and monotonic abstraction concludes the Section.

The methodology we exploit to check safety of a program $\mathcal{P} = (\mathbf{v}, l_I, l_E, T)$ is *backward search*: we successively explore, through symbolic representation, all states leading to the error location l_E in one step, then in two steps, in three steps, etc. until either we find a fixpoint or until we reach l_I. To do this properly, it is convenient to build a tree: the tree has arcs labeled by transitions and nodes labeled by formulæ over $\mathbf{v}$. Leaves of the tree might be marked 'checked', 'unchecked' or 'covered'. The tree is built according to the following non-deterministic rules.

Backward Search

Initialization: a single node tree labeled by $pc = l_E$ and is marked 'unchecked'.

Check: pick an unchecked leaf L labeled with K. If $K \wedge pc = l_I$ is satisfiable ('safety test'), exit and return unsafe. If it is not satisfiable, check whether there is a set S of uncovered nodes such that (i) $L \not\in S$ and (ii) K is inconsistent with the conjunction of the negations of the formulæ labeling the nodes in S ('fixpoint check'). If it is so, mark L as 'covered' (by S). Otherwise, mark L as 'checked'.

Expansion: pick a checked leaf L labeled with K. For each transition $\tau_i \in T$, add a new leaf below L labeled with $Pre(\tau_i, L)$ and marked as 'unchecked'. The arc between L and the new leaf is labeled with τ_i.

Safety Exit: if all leaves are covered, exit and return safe.

The algorithm may not terminate (this is unavoidable by well-known undecidability results). Its correctness depends on the possibility of discharging safety tests with complete algorithms. By Proposition 1, if transitions are ground- or Σ^0_1-assignments, completeness of safety tests arising during the backward reachability procedure is guaranteed by the fact that satisfiability of Σ^0_1-formulæ is decidable. For fixpoint tests, sound but incomplete algorithms may compromise termination, but not correctness of the answer; hence for fixpoint tests, we can adopt incomplete pragmatic algorithms (e.g. if in fixpoint tests we need to test satisfiability of Σ^0_2-sentences, the obvious strategy is to Skolemize existentially quantified variables and to instantiate the universally quantified ones over sets of terms chosen according to suitable heuristics). To sum up, we have:

Proposition 2. *The above* Backward Search *procedure is partially correct for programs whose transitions are Σ^0_1-assignments, i.e., when the procedure terminates it gives a correct information about the safety of the input program.*

Divergence phenomena are usually not due to incomplete algorithms for fixpoint tests (in fact, divergence persists even in cases where fixpoint tests are precise).

Example 1. Consider the program in Fig. 1(b): it reverses the content of the array I into O. In our formalism, it is represented by the following transitions[4]:

$$\begin{aligned}
\tau_1 &\equiv \ \mathtt{pc} = 1 \wedge \mathtt{pc}' = 2 \wedge \mathtt{c}' = 0 \\
\tau_2 &\equiv \ \mathtt{pc} = 2 \wedge \mathtt{c} \neq N+1 \wedge \mathtt{c}' = \mathtt{c}+1 \wedge O' = wr(O, \mathtt{c}, I(N-\mathtt{c})) \\
\tau_3 &\equiv \ \mathtt{pc} = 2 \wedge \mathtt{c} = N+1 \wedge \mathtt{pc}' = 3 \\
\tau_4 &\equiv \ \mathtt{pc} = 3 \wedge \exists z_1 \geq 0, z_2 \geq 0 \ (z_1 + z_2 = N \wedge I(z_1) \neq O(z_2) \ ) \wedge \mathtt{pc}' = 4.
\end{aligned}$$

Notice that τ_1, τ_2, τ_3 all are ground assignments; only τ_4 (that translates the error condition) is a Σ_1^0-assignment. If we apply our tree generation procedure, we get an infinite branch, whose nodes - after routine simplifications - are labeled as follows

$$\begin{aligned}
&\ldots \\
(K_i) \ \ &\mathtt{pc} = 2 \wedge \exists z_1, z_2 \ \psi(z_1, z_2) \wedge \mathtt{c} = N - i \wedge z_2 \neq N \wedge \cdots \wedge z_2 \neq N - i \\
&\ldots
\end{aligned}$$

where $\psi(z_1, z_2)$ stands for $z_1 \geq 0 \wedge z_2 \geq 0 \wedge z_1 + z_2 = N \wedge I(z_1) \neq O(z_2)$. □

As demonstrated by the above example, a divergence source comes from the fact that we are unable to represent *in one shot* the effect of executing finitely many times a given sequence of transitions. Acceleration can solve this problem.

Definition 1. *The n-th composition of a transition $\tau(\mathbf{v}, \mathbf{v}')$ with itself is recursively defined by $\tau^1 := \tau$ and $\tau^{n+1} := \tau \circ \tau^n$. The* acceleration *$\tau^+$ of τ is $\bigvee_{n \geq 1} \tau^n$.*

In general, acceleration requires a logic supporting infinite disjunctions. Notable exceptions are witnessed by Theorem 1 (Section 6). For now we focus on examples where accelerations yield Σ_2^0-assignments starting from ground assignments.

Example 2. Recall transition τ_2 from the running example.

$$\tau_2 \equiv \ \mathtt{pc} = 2 \wedge \mathtt{c} \neq N+1 \wedge pc' = 2 \wedge \mathtt{c}' = \mathtt{c}+1 \wedge I' = I \wedge O' = wr(O, \mathtt{c}, I(N-\mathtt{c}))$$

(here we displayed identical updates for completeness). Notice that the variable pc is left unchanged in this transition (this is essential, otherwise the acceleration gives an inconsistent transition that can never fire). If we accelerate it, we get the Σ_2^0-assignment[5]

$$\exists n > 0 \left(\begin{array}{c} \mathtt{pc} = 2 \ \wedge \ \forall j \ (\mathtt{c} \leq j < \mathtt{c}+n \rightarrow j \neq N+1) \ \wedge \ \mathtt{c}' = \mathtt{c}+n \ \wedge \\ \wedge \ \mathtt{pc}' = 2 \ \wedge \ O' = \lambda j \ (\mathtt{if} \ \mathtt{c} \leq j < \mathtt{c}+n \ \mathtt{then} \ I(N-j) \ \mathtt{else} \ O(j)) \end{array} \right) \quad (4)$$

□

In presence of these accelerated Σ_2^0-assignments, Backward Search can produce problematic Σ_2^0-sentences (see Proposition 1 above) which cannot be handled precisely by existing solvers. As a solution to this problem we propose applying to such sentences a suitable abstraction, namely *monotonic abstraction.*

[4] For readability, we omit identical updates like $I' = I$, etc. Notice that we have $l_I = 1$ and $l_E = 4$.

[5] This Σ_2^0-assignment can be automatically computed using procedures outlined in the proof of Theorem 1.

Definition 2. *Let $\psi :\equiv \exists \underline{i}\, \forall \underline{j}.\ \phi(\underline{i}, \underline{j}, \mathbf{a}, \mathbf{c}, pc)$ be a Σ^0_2-sentence and let $\mathcal{S}$ be a finite set of terms of the form $t(\underline{i}, \mathbf{v})$. The* monotonic $\mathcal{S}$-approximation of ψ *is the Σ^0_1-sentence*

$$\exists \underline{i} \bigwedge_{\sigma:\underline{j}\to\mathcal{S}} \phi(\underline{i}, \underline{j}\sigma/\underline{j}, \mathbf{a}, \mathbf{c}, pc) \tag{5}$$

(here $\underline{j}\sigma$ is the tuple of terms $\sigma(j_1), \dots, \sigma(j_n)$, where $\underline{j} = j_1, \dots, j_n,$).

By Definition 2, universally quantified variables are *eliminated through instantiation*; the larger the set $\mathcal{S}$ is, the better approximation you get. In practice, the natural choices for $\mathcal{S}$ are $\underline{i}$ or the set of terms of the kind $t(\underline{i}, \mathbf{v})$ occurring in ψ We adopted the former choice in our implementation. As a result of replacing Σ^0_2-sentences by their monotonic approximation, spurious unsafe traces might occur. However, **those can be disregarded if accelerated transitions contribute to their generation**. This is because if $\mathcal{P}$ is unsafe, then unsafety can be discovered without appealing to accelerated transitions.

To integrate monotonic abstraction, the above Backward Search procedure is modified as follows. In a Preprocessing step, we add some accelerated transitions of the kind $(\tau_1 \circ \cdots \circ \tau_n)^+$ to T. These transitions can be found by inspecting cycles in the control flow graph of the program and accelerating them following the procedure described in Sections 5, 6. The natural cycles to inspect are those corresponding to loop branches in the source code. It should be noticed, however, that identifying the good cycles to accelerate is subject to specific heuristics that deserve separate investigation in case the program has infinitely many cycles (Choosing cycles from branches of innermost loops is the simplest example of such heuristics and the one we implemented).

After this extra preprocessing step, the remaining instructions are left unchanged, with the exception of Check that is modified as follows:

Check': pick an unchecked leaf L labeled by a formula K. If K is a Σ^0_2-sentence, choose a suitable $\mathcal{S}$ and replace K by its monotonic $\mathcal{S}$-abstraction K'. If $K' \wedge pc = l_I$ is inconsistent, mark L as 'covered' or 'checked' according to the outcome of the fixpoint check, as was done in the original Check. If $K' \wedge pc = l_I$ is satisfiable, analyze the path from the root to L. If no accelerated transition τ^+ is found in it return unsafe, otherwise remove the sub-tree D from the target of τ^+ to the leaves. Each node N covered by a node in D will be flagged as 'unchecked' (to make it eligible in future for the Expansion instruction).

The new procedure will be referred as Backward Search'. It is quite straightforward to see that Proposition 2 still applies to the modified algorithm. Notice that, although termination cannot be ensured (given well-known undecidability results), spurious traces containing approximated accelerated transitions cannot be produced again and again: when the sub-tree D from the target node v of τ^+ is removed by Check', the node v is not a leaf (the arcs labeled by the transitions τ are still there), hence it cannot be expanded anymore according to the Expansion instruction.

Example 3. Let us again consider the running example and demonstrate how acceleration and monotonic abstraction work. In the preprocessing step, we add the accelerated transition τ_2^+ given by (4) to the transitions we already have. After having computed $(K') \equiv \mathit{Pre}(\tau_4, K), (K'') \equiv \mathit{Pre}(\tau_3, K')$, we compute $(\tilde{K}) \equiv \mathit{Pre}(\tau_2^+, K'')$ and get

$$\exists n > 0 \, \exists z_1, z_2 \begin{pmatrix} \texttt{pc} = 2 \ \wedge \ \forall j \ (\texttt{c} \leq j < \texttt{c}{+}n \to j \neq N{+}1) \ \wedge \\ \wedge \ \texttt{c}{+}n = N{+}1 \ \wedge z_1 \geq 0 \ \wedge \ z_2 \geq 0 \ \wedge z_1 + z_2 = N \ \wedge \\ \wedge \ I(z_1) \neq \lambda j \ (\texttt{if c} \leq j < \texttt{c} + n \ \texttt{then} \ I(N{-}j) \ \texttt{else} \ O(j))(z_2) \end{pmatrix}$$

We approximate using the set of terms $\mathcal{S} = \{z_1, z_2, n\}$. After simplifications we get $\exists z_1, z_2 \, (\texttt{pc} = 2 \ \wedge \ \texttt{c} \leq N \ \wedge z_1 \geq 0 \ \wedge \ z_2 \geq 0 \ \wedge \ z_1 + z_2 = N \ \wedge \ O(z_2) \neq I(z_1) \ \wedge \ \texttt{c} > z_2)$ Generating this formula is enough to stop divergence. □

Notice that in the computations of the above example we eventually succeeded in eliminating the extra quantifier $\exists n$ introduced by the accelerated transition. This is not always possible: sometimes in fact, to get the good invariant one needs more quantified variables than those occurring in the annotated program and accelerated transitions might be the way of getting such additional quantified variables.

5 Iterators

This Section introduces *iterators* and *selectors*, two main ingredients used to supply a useful format to compute accelerated transitions. Iterators are meant to formalize the notion of a counter scanning the indexes of an array: the most simple iterators are increments and decrements, but one may also build more complex ones for different scans, like in binary search. We give their formal definition and then we supply some examples. We need to handle tuples of terms because we want to consider the case in which we deal with different arrays with possibly different scanning variables. Given a m-tuple of terms

$$\mathbf{u}(\underline{x}) \ := \ u_1(x_1, \ldots, x_m), \ldots, u_m(x_1, \ldots, x_m) \tag{6}$$

containing the m variables $\underline{x} = x_1, \ldots, x_m$, we indicate with $\mathbf{u}^n$ the term expressing the n-times composition of (the function denoted by) $\mathbf{u}$ with itself. Formally, we have $\mathbf{u}^0(\underline{x}) := \underline{x}$ and

$$\mathbf{u}^{n+1}(\underline{x}) \ := \ u_1(\mathbf{u}^n(\underline{x})), \ldots, u_m(\mathbf{u}^n(\underline{x})) \ .$$

Definition 3. *A tuple of terms* $\mathbf{u}$ *like* (6) *is said to be an* iterator *iff there exists an* m*-tuple of* $m+1$*-ary terms* $\mathbf{u}^*(\underline{x}, y) \ := \ u_1^*(x_1, \ldots, x_m, y), \ldots, u_m^*(x_1, \ldots, x_m, y)$ *such that for any natural number* n *it happens that the formula*

$$\mathbf{u}^n(\underline{x}) = \mathbf{u}^*(\underline{x}, \bar{n}) \tag{7}$$

is valid.[6] *Given an iterator* $\mathbf{u}$ *as above, we say that an* m*-ary term* $\kappa(x_1, \ldots, x_m)$ *is a* selector *for* $\mathbf{u}$ *iff there is an* $m+1$*-ary term* $\iota(x_1, \ldots, x_m, y)$ *yielding the validity of the formula*

$$z = \kappa(\mathbf{u}^*(\underline{x}, y)) \to y = \iota(\underline{x}, z) \ \ . \tag{8}$$

[6] Recall that $\bar{n}$ is the numeral of n, i.e. it is $s^n(0)$.

The meaning of condition (8) is that, once the input $\underline{x}$ and the selected output z are known, it is possible to identify uniquely (through ι) the number of iterations y that are needed to get z by applying κ to $\mathbf{u}^*(\underline{x}, y)$. The term κ is a selector function that selects (and possibly modifies) one of the $\mathbf{u}$; in most applications (though not always) κ is a projection, represented as a variable x_i (for $1 \leq i \leq m$), so that $\kappa(\mathbf{u}^*(\underline{x}, y))$ is just the i-th component $u_i^*(\underline{x}, y)$ of the tuple of terms $\mathbf{u}^*(\underline{x}, y)$. In these cases, the formula (8) reads as

$$z = u_i^*(\underline{x}, y) \rightarrow y = \iota(\underline{x}, z) \ . \tag{9}$$

Example 4. The canonical example is when we have $m = 1$ and $\mathbf{u} := u_1(x_1) := x_1 + 1$; this is an iterator with $u_1^*(x_1, y) := x_1 + y$; as a selector, we can take $\kappa(x_1) := x_1$ and $\iota(x_1, z) := z - x_1$. □

Example 5. The previous example can be modified, by choosing $\mathbf{u}$ to be $x_1 + \bar{n}$, for some integer $n \neq 0$: then we have $u^*(x_1, y) := x_1 + n * y$, $\kappa(x_1) := x_1$, and $\iota(x_1, z) = (z - x_1)//n$ where $//$ is integer division (recall that integer division by a given n is definable in Presburger arithmetic). □

Example 6. If we move to more expressive arithmetic theories, like Primitive Recursive Arithmetic (where we have a symbol for every primitive recursive function), we can get much more examples. As an example with $m > 1$, we can take $\mathbf{u} := x_1 + x_2, x_2$ and get $u_1^*(x_1, x_2, y) = x_1 + y * x_2$, $u_2^*(x_1, x_2, y) = x_2$. Here a selector is for instance $\kappa_1(x_1, x_2) := \bar{7} + x_1$, $\iota(x_1, x_2, z) := (z - x_1 - \bar{7})//x_2$. □

6 Accelerating Local Ground Assignments

Back to our program $\mathcal{P} = (\mathbf{v}, l_I, l_E, T)$, we look for conditions on transitions from T allowing to accelerate them via a Σ_2^0-assignment. Given an iterator $\mathbf{u}(\underline{x})$, a *selector assignment* for $\mathbf{a} := a_1, \ldots, a_s$ (relative to $\mathbf{u}$) is a tuple of selectors $\kappa := \kappa_1, \ldots, \kappa_s$ for $\mathbf{u}$. Intuitively, the components of the tuple are meant to indicate the scanners of the arrays $\mathbf{a}$ and as such might not be distinct (although, of course, just *one* selector is assigned to each array). A formula ψ (resp. a term t) is said to be *purely arithmetical* over a finite set of terms V iff it is obtained from a formula (resp. a term) *not containing the extra free function symbols* $\mathbf{a}, \mathbf{c}$ by replacing some free variables in it by terms from V. Let $\mathbf{v} = v_1, \ldots, v_s$ and $\mathbf{w} = w_1, \ldots, w_s$ be s-tuples of terms; below $wr(\mathbf{a}, \mathbf{v}, \mathbf{w})$ and $\mathbf{a}(\mathbf{v})$ indicate the tuples $wr(a_1, v_1, w_1), \ldots, wr(a_s, v_s, w_s)$ and $a_1(v_1), \ldots, a_s(v_s)$, respectively (recall from Section 3 that $s = |\mathbf{a}|$).

Definition 4. *A* local ground assignment *is a ground assignment of the form*

$$pc = l \wedge \phi_L(\mathbf{c}, \mathbf{a}) \wedge pc' = l \wedge \mathbf{a}' = wr(\mathbf{a}, \kappa(\tilde{\mathbf{c}}), \mathbf{t}(\mathbf{c}, \mathbf{a})) \wedge \tilde{\mathbf{c}}' = \mathbf{u}(\tilde{\mathbf{c}}) \wedge \mathbf{d}' = \mathbf{d} \tag{10}$$

where (i) $\mathbf{c} = \tilde{\mathbf{c}}, \mathbf{d}$*;* (ii) $\mathbf{u} = u_1, \ldots, u_{|\tilde{\mathbf{c}}|}$ *is an iterator;* (iii) *the terms* κ *are a selector assignment for* $\mathbf{a}$ *relative to* $\mathbf{u}$*;* (iv) *the formula* $\phi_L(\mathbf{c}, \mathbf{a})$ *and the terms* $\mathbf{t}(\mathbf{c}, \mathbf{a})$ *are purely arithmetical over the set of terms* $\{\mathbf{c}, \mathbf{a}(\kappa(\tilde{\mathbf{c}}))\} \cup \{a_i(d_j)\}_{1 \leq i \leq s, 1 \leq j \leq |\mathbf{d}|}$*;* (v) *the guard* ϕ_L *contains the conjuncts* $\kappa_i(\tilde{\mathbf{c}}) \neq d_j$*, for* $1 \leq i \leq s$ *and* $1 \leq j \leq |\mathbf{d}|$.

Thus in a local ground assignment, there are various restrictions: (a) the numerical variables are split into 'idle' variables $\mathbf{d}$ and variables $\tilde{\mathbf{c}}$ subject to update via an iterator $\mathbf{u}$; (b) the program counter is not modified; (c) the guard does not depend on the values of the a_i at cells different from $\kappa_i(\tilde{\mathbf{c}}), \mathbf{d}$; (d) the update of the $\mathbf{a}$ are simultaneous writing operations modifying only the entries $\kappa(\tilde{\mathbf{c}})$. Thus, the assignment is local and the relevant modifications it makes are determined by the selectors locations. The 'idle' variables $\mathbf{d}$ are useful to accelerate branches of nested loops; the inequalities mentioned in (v) are automatically generated by making case distinctions in assignment guards.

Example 7. For our running example, we show that transition τ_2 (the one we want to accelerate) is a local ground assignment. We have $\mathbf{d} = \emptyset$ and $\tilde{\mathbf{c}} = \mathtt{c}$ and $\mathbf{a} = I, O$. The counter $\mathtt{c}$ is incremented by 1 at each application of τ_2. Thus, our iterator is $\mathbf{u} := x_1 + 1$ and the selector assignment assigns $\kappa_1 := N - x_1$ to I and $\kappa_2 := x_1$ to O. In this way, I is modified (identically) at $N - \mathtt{c}$ via $I' = wr(I, N - \mathtt{c}, I(N - \mathtt{c}))$ and O is modified at $\mathtt{c}$ via $O' = wr(O, \mathtt{c}, I(N - \mathtt{c}))$. The guard τ_2 is $\mathtt{c} \neq N + 1$. Since the formula $\mathtt{c} \neq N + 1$ and the term $I(N - \mathtt{c})$ are purely arithmetical over $\{\mathtt{c}, I(N - \mathtt{c}), O(\mathtt{c})\}$, we conclude that τ_2 is local. □

Theorem 1. *If τ is a local ground assignment, then τ^+ is a Σ^0_2-assignment.*

Proof. (Sketch, see [6] for full details). Let us fix the local ground assignment (10); let $\mathbf{a}[\mathbf{d}]$ indicate the $s * |\mathbf{d}|$-tuple of terms $\{a_i(d_j)\}_{1 \leq i \leq s, 1 \leq j \leq |\mathbf{d}|}$; since ϕ_L and $\mathbf{t} := t_1, \ldots, t_s$ are purely arithmetical over $\{\tilde{\mathbf{c}}, \mathbf{d}, \mathbf{a}(\kappa(\tilde{\mathbf{c}})), \mathbf{a}[\mathbf{d}]\}$, we have that they can be written as $\tilde{\phi}_L(\tilde{\mathbf{c}}, \mathbf{d}, \mathbf{a}(\kappa(\tilde{\mathbf{c}})), \mathbf{a}[\mathbf{d}])$, $\tilde{\mathbf{t}}(\tilde{\mathbf{c}}, \mathbf{d}, \mathbf{a}(\kappa(\tilde{\mathbf{c}})), \mathbf{a}[\mathbf{d}])$, respectively, where $\tilde{\phi}_L, \tilde{\mathbf{t}}$ do not contain occurrences of the free function and constant symbols $\mathbf{a}, \mathbf{c}$. The transition τ^+ can be expressed as a Σ^0_2-assignment by

$$\exists y > 0 \left(\begin{array}{c} \forall z \; (0 \leq z < y \to \tilde{\phi}_L(\mathbf{u}^*(\tilde{\mathbf{c}}, z), \mathbf{d}, \mathbf{a}(\kappa(\mathbf{u}^*(\mathbf{c}, z))), \mathbf{a}[\mathbf{d}]) \wedge \mathbf{d}' = \mathbf{d} \wedge \\ \wedge \; pc = l \;\wedge\; pc' = l \;\wedge\; \tilde{\mathbf{c}}' = \mathbf{u}^*(\tilde{\mathbf{c}}, y) \;\wedge\; \mathbf{a}' = \lambda j.\, F(\mathbf{c}, \mathbf{a}, y, j) \end{array} \right)$$

where the tuple $F = F_1, \ldots, F_s$ of definable functions is given by

$$\begin{array}{rcl} F_h(\mathbf{c}, \mathbf{a}, y, j) & = & \lambda j. \;\; \mathtt{if} \;\; 0 \leq \iota_h(\tilde{\mathbf{c}}, j) < y \;\wedge j = \kappa_h(\mathbf{u}^*(\mathbf{c}, \iota_h(\tilde{\mathbf{c}}, j))) \;\mathtt{then} \\ & & \tilde{t}_h(\mathbf{u}^*(\tilde{\mathbf{c}}, \iota_h(\tilde{\mathbf{c}}, j)), \mathbf{d}, \mathbf{a}(\kappa(\mathbf{u}^*(\tilde{\mathbf{c}}, \iota_h(\tilde{\mathbf{c}}, j)))), \mathbf{a}[\mathbf{d}]) \;\mathtt{else}\; a_h[j] \end{array}$$

for $h = 1, \ldots, s$ (here $\iota_1, \ldots, \iota_s$ are the terms corresponding to $\kappa_1, \ldots, \kappa_s$ according to the definition of a selector for the iterator $\mathbf{u}$). $\dashv$

We point out that the effective use of Theorem 1 relies on the implementation of a repository of iterators and selectors and of algorithms recognizing them. The larger the repository is, the more possibilities the model checker has to exploit the full power of acceleration.

In most applications it is sufficient to consider accelerated transitions of the canonical form of Example 4. Let us examine in details this special case; here $\mathbf{c}$ is a single counter $\mathtt{c}$ that is incremented by one (otherwise said, the iterator is $x_1 + 1$) and the selector assignment is trivial, namely it is just x_1. We call these local ground assignments *simple*. Thus, a simple local ground assignment has the form

$$pc = l \ \wedge \ \phi_L(\mathtt{c}, \mathbf{a}) \ \wedge \ pc' = l \wedge \mathtt{c}' = \mathtt{c} + 1 \ \wedge \ \mathbf{a}' = wr(\mathbf{a}, \mathtt{c}, \mathbf{t}(\mathtt{c}, \mathbf{a})) \tag{11}$$

where the first occurrence of $\mathtt{c}$ in $wr(\mathbf{a}, \mathtt{c}, \mathbf{t}(\mathtt{c}, \mathbf{a}))$ stands in fact for an s-tuple of terms all identical to $\mathtt{c}$, and where $\phi_L, \mathbf{t}$ are purely arithmetical over the terms $\mathtt{c}$, $a_1[\mathtt{c}], \ldots, a_s[\mathtt{c}]$. The accelerated transition computed in the proof of Theorem 1 for (11) can be rewritten as follows:

$$\exists k \left(\begin{array}{c} k > 0 \ \wedge \ pc = l \ \wedge \ \forall j \ (\mathtt{c} \leq j < \mathtt{c} + k \rightarrow \phi_L(j, \mathbf{a})) \ \wedge \ pc' = l \ \wedge \\ \wedge \ \mathtt{c}' = \mathtt{c} + k \ \wedge \ \mathbf{a}' = \lambda j. \ (\mathtt{if}\ \mathtt{c} \leq j < \mathtt{c} + k \ \mathtt{then}\ \mathbf{t}(j, \mathbf{a}) \ \mathtt{else}\ \mathbf{a}[j]) \end{array} \right) \tag{12}$$

A slight extension of the notion of a simple assignment leads to a further subclass of local ground assignments useful to accelerated branches of nested loops (see [6] for more details).

7 Experimental Evaluation

We implemented the algorithm described in Section 4–6 as a preprocessing module inside the MCMT model checker [24]. To perform a feasibility study, we intentionally focused our implementation on simple and simple+ local ground assignments. For a thorough and unbiased evaluation we compared/combined the new technique with an abstraction algorithm suited for array programs [3] implemented in the same tool. This section describes benchmarks and discusses experimental results. A clear outcome from our experiments is that abstraction/refinement and acceleration techniques can be gainfully combined.

Benchmarks. We evaluated the new algorithm on 55 programs with arrays, each annotated with an assertion. We considered only quantifier-free or ∀-assertions. Our set of benchmarks comprises programs used to evaluate the Lazy Abstraction with Interpolation for Arrays framework [4] and other focused benchmarks where abstraction diverges. These are problems involving array manipulations such as copying, comparing, searching, sorting, initializing, testing, etc. About one third of the programs contain bugs.[7]

Evaluation. Experiments have been run on a machine equipped with a i7@2.66 GHz CPU and 4GB of RAM running OS X. Time limit for each experiment has been set to 60 seconds. We run MCMT with four different configurations:

- Backward Search - MCMT executes the procedure described at the beginning of Section 4.
- Abstraction - MCMT integrates the backward reachability algorithm with the abstraction/refinement loop [3].

[7] The set of benchmarks can be downloaded from `http://www.inf.usi.ch/phd/alberti/prj/acc`; the tool set MCMT is available at `http://users.mat.unimi.it/users/ghilardi/mcmt/`.

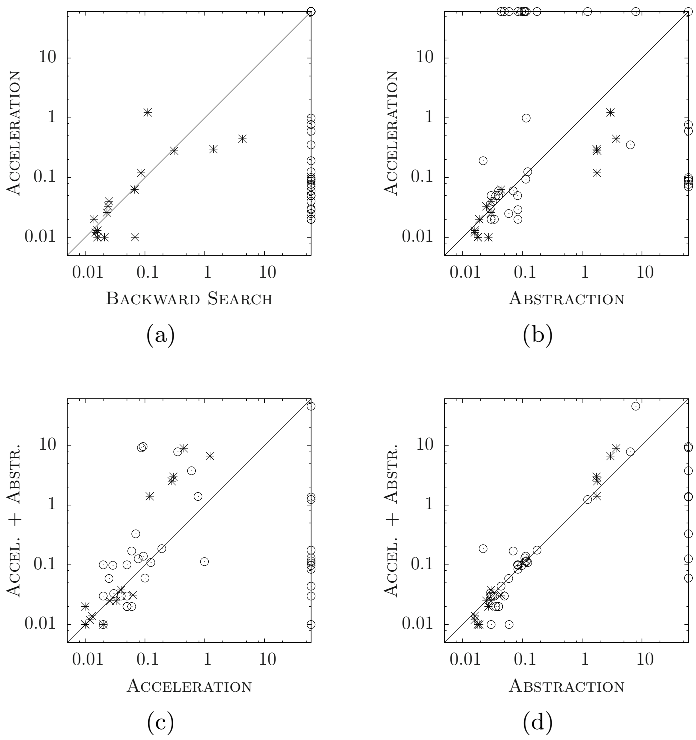


Fig. 2. Comparison of time for different options of Backward Search. Stars and circles represent buggy and correct programs respectively.

- Acceleration - The transition system is pre-processed in order to compute accelerated transitions (when it is possible) and then the Backward Search' procedure is executed.
- Accel. + Abstr. - This configuration enables both the preprocessing step in charge of computing accelerated transitions and the abstraction/refinement engine on the top of the Backward Search' procedure.

In summary, the comparative analysis of timings presented in Fig.2 confirms that acceleration indeed helps to avoid divergence for problematic programs where abstraction fails. The first comparison (Fig.2(a)) highlights the benefits of using acceleration: Backward Search diverges on all 39 safe instances. Acceleration stops divergence in 23 cases, and moreover the overhead introduced by the preprocessing step does not affect unsafe instances. Fig.2(b) shows that acceleration and abstraction are two complementary techniques, since mcmt times out in both cases but for two different sets of programs. Fig.2(c) and Fig.2(d) attest that acceleration and abstraction/refinement techniques mutually benefit from each other: with both techniques mcmt solves all the 55 benchmarks.

8 Conclusion and Future Work

We identified a class of transition relations involving array updates that can be accelerated, showed how it is possible to compute accelerated transition and described a solution for dealing with universal quantifiers arising from the acceleration process. Our paper lays theoretical foundations for this interesting research topic and confirms by our prototype experiments on challenging benchmarks its advantages over stand-alone verification approaches since it is able to solve problems on which other techniques fail to converge.

As future directions, a challenging task is to enlarge the definability result of Theorem 1 to cover classes of transitions modeling more and more loop branches arising from concrete programs. In addition, one may want to consider more sophisticated strategies for instantiation in order to support acceleration. Increasing the approximation-defining sets $\mathcal{S}$ or handling Σ_2^0-sentences when they belong to decidable fragments [16, 23] may lead to further improvements.

Acknowledgements. The work of the first author was supported by the Hasler Foundation under project 09047 and the one of the second by Italian Ministry of Education, University and Research (MIUR) under the PRIN 2010-2011 project "Logical Methods for Information Management".

References

1. Abdulla, P.A., Delzanno, G., Ben Henda, N., Rezine, A.: Regular model checking without transducers (On efficient verification of parameterized systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
2. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
3. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy Abstraction with Interpolants for Arrays. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 46–61. Springer, Heidelberg (2012)
4. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 679–685. Springer, Heidelberg (2012)
5. Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., Rossi, G.P.: Universal Guards, Relativization of Quantifiers, and Failure Models in Model Checking Modulo Theories. JSAT, 29–61 (2012)
6. Alberti, F., Ghilardi, S., Sharygina, N.: Tackling divergence: abstraction and acceleration in array programs. Technical Report 2012/01, University of Lugano (October 2012)
7. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
8. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0 (2010), http://www.smt-lib.org
9. Behrmann, G., Bengtsson, J., David, A., Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL implementation secrets. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 3–22. Springer, Heidelberg (2002)

10. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI, pp. 300–309 (2007)
11. Borralleras, C., Lucas, S., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: SAT modulo linear arithmetic for solving polynomial constraints. J. Autom. Reasoning 48(1), 107–131 (2012)
12. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 337–351. Springer, Heidelberg (2009)
13. Bozga, M., Habermehl, P., Iosif, R., Konečný, F., Vojnar, T.: Automatic verification of integer array programs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 157–172. Springer, Heidelberg (2009)
14. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010)
15. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. Fundam. Inform. 91(2), 275–303 (2009)
16. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
17. Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating interpolation-based model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 428–442. Springer, Heidelberg (2008)
18. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. ACM Trans. Comput. Log. 12(1), 7 (2010)
19. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
20. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 268–279. Springer, Heidelberg (1998)
21. Finkel, A., Leroux, J.: How to compose presburger-accelerations: Applications to broadcast protocols. In: Agrawal, M., Seth, A.K. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 145–156. Springer, Heidelberg (2002)
22. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, pp. 191–202 (2002)
23. 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)
24. Ghilardi, S., Ranise, S.: MCMT: A Model Checker Modulo Theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 22–29. Springer, Heidelberg (2010)
25. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
26. Hendriks, M., Larsen, K.G.: Exact acceleration of real-time model checking. Electr. Notes Theor. Comput. Sci. 65(6), 120–139 (2002)
27. Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 187–202. Springer, Heidelberg (2012)
28. Jhala, R., McMillan, K.L.: Array Abstractions from Proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)

29. Kovács, L., Voronkov, A.: Interpolation and Symbol Elimination. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 199–213. Springer, Heidelberg (2009)
30. Larraz, D., Rodríguez-Carbonell, E., Rubio, A.: SMT-based array invariant generation. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 169–188. Springer, Heidelberg (2013)
31. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
32. McMillan, K.L.: Quantified Invariant Generation Using an Interpolating Saturation Prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
33. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transaction on Programming Languages and Systems 1(2), 245–257 (1979)
34. Seghir, M.N., Podelski, A., Wies, T.: Abstraction Refinement for Quantified Array Assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009)
35. Srivastava, S., Gulwani, S.: Program Verification using Templates over Predicate Abstraction. In: PLDI (2009)
36. Tinelli, C., Harandi, M.T.: A new correctness proof of the Nelson-Oppen combination procedure. In: Proc. of FroCoS 1996, pp. 103–119. Kluwer (1996)

Verification of Composed Array-Based Systems with Applications to Security-Aware Workflows

Clara Bertolissi[1,2] and Silvio Ranise[2]

[1] LIF-CNRS, UMR 7279 & AMU, Marseille, France
[2] FBK (Fondazione Bruno Kessler), Trento, Italy

Abstract. We introduce a class of symbolic transition systems capable of representing collections of security-aware workflows and we study the verification of reachability properties of such systems. More precisely, we define *composed* array-based systems as an extension of array-based systems in which array variables are indexed over more than one type. For an application relevant sub-class of these systems we show how to mechanize a symbolic backward reachability procedure by modularly re-using the techniques developed for array-based systems. Finally, and most importantly, we find sufficient conditions for the termination of the procedure and we apply this result to derive the decidability of the reachability problems of two important classes of security-aware workflow systems.

1 Introduction

Many E-services, such as business processes, are modelled as workflows, which often need to comply with authorization policies. A workflow specifies a collection of tasks to be executed by users, together with a set of causal dependencies between tasks. The design of E-services is a difficult and error prone activity as a single service may comprise several, concurrently executing, workflow instances. Design errors can thus arise from interleaved access over shared data or synchronization between different workflow instances. The situation is complicated by the presence of authorization constraints such as Bound of Duties (BoD), i.e. the same user should execute two tasks, or Separation of Duties (SoD), i.e. distinct users must execute the two tasks. Following [2], we call "security-aware" the workflows that involve this kind of constraints. This may give rise to situations in which a user should and, at the same time, is prohibited to execute certain tasks or that a workflow instance cannot terminate without violating one or more SoD (or BoD) constraints, despite the fact that the users are entitled to execute those tasks. When authorization constraints may be imposed not only within a workflow instance but also among two or more instances (this is useful to reduce the risk of frauds or for workload or resource re-distribution) [15], understanding the consequences of the interplay between concurrent execution of workflow instances and authorization constraints becomes very difficult if possible at all. For these reasons, automated verification techniques for security-aware workflow systems are of paramount importance to help in the design of E-services.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 40–55, 2013.

In this paper, we introduce a class of symbolic transition systems capable of modelling a finite but unknown number of security-aware workflow instances or of users responsible to execute them and study the verification of reachability properties of such systems. Our verification technique, based on symbolic backward reachability, is capable of verifying properties for any number of workflow instances and any number of users. When the technique detects that there is a sequence of transitions from the initial state to one satisfying the reachability property, it returns a concrete configuration—i.e. the numbers of workflow instances and users—for which this is the case. On the contrary, when the technique concludes that no such sequence of transitions exists, it does so for any configuration, i.e. regardless of the number of workflow instances and users.

We develop our results in the framework of Model Checking Modulo Theories [12]. In particular, we introduce *composed array-based systems* as an extension of array-based systems (Section 3). The main difference between the two notions is in terms of the indexes used to dereference array variables. In array-based systems, there is only one type for indexes (formalized as a theory) and it is not possible to express transitions that depend on both workflow instances and users as it is the case of security-aware workflows. Instead, in composed array-based systems, indexes may belong to several types. A simple example to illustrate the adequacy of composed array-based systems for the specification of a security-aware workflow system adapted from [6] is shown in Section 3.1. It is also used to motivate the study of a particular class of composed array-based systems, called *n-components array-based systems* (Section 3.2), in which the symbolic representation of a set of states is obtained by a class of formulae, called *component formulae*, in which only array variables whose indexes are of a single type may occur. The symbolic representation of transitions is obtained by conjoining component formulae.

For n-components array-based systems, we show how to mechanize a symbolic backward reachability procedure by using Satisfiability Modulo Theories (SMT) solving (Section 4) and modularly re-using the techniques developed for array-based systems in [12]. More importantly (in Section 5), we find sufficient conditions for the termination of the backward reachability procedure by lifting the approach in [12] that uses well-quasi-orderings (wqos) [10]. The idea is to modularly re-use each wqo that can be defined over a component formula to define an (extension) ordering over their disjunctions, that is guaranteed to be a wqo by the same argument that is at the core of the proof of Dickson's Lemma (see, e.g., [10]).

Finally (Section 6), we apply our termination result for n-components array-based systems to show the decidability of two important classes of security-aware workflow systems.

2 Formal Preliminaries

For making this paper self-contained, we recall some definitions from [12]. We assume the usual syntactic (e.g., signature, variable, term, atom, literal, and formula) and semantic (e.g., structure, sub-structure, assignment, truth, satisfiability, and validity) notions of many-sorted first-order logic (see, e.g., [8]). The equality symbol $=$ is included in all signatures considered below. If $\mathcal{M}$ is a structure for a signature Σ (briefly, a Σ-structure), we denote by $S^{\mathcal{M}}, f^{\mathcal{M}}, P^{\mathcal{M}}, \ldots$ the interpretation in $\mathcal{M}$ of the sort S, the function symbol f, the predicate symbol P, etc. If Σ_0 is a sub-signature of Σ, the *restriction* of $\mathcal{M}$ to Σ_0, denoted $\mathcal{M}_{|\Sigma_0}$, is the structure resulting from $\mathcal{M}$ by forgetting about the interpretation of the sorts, function and predicate symbols that are not in Σ_0. A Σ-structure $\mathcal{N}$ is a *sub-structure* of a Σ-structure $\mathcal{M}$ iff the domain of $\mathcal{N}$ is contained in the domain of $\mathcal{M}$ and the interpretations of the symbols of Σ in $\mathcal{N}$ are restrictions of the interpretations of these symbols in $\mathcal{M}$. A class $\mathcal{C}$ of Σ-structures is *closed under sub-structures* iff for every structure $\mathcal{M} \in \mathcal{C}$, if $\mathcal{N}$ is a substructure of $\mathcal{M}$ then $\mathcal{N} \in \mathcal{C}$.

A *theory* T is a pair $(\Sigma, \mathcal{C})$, where Σ is a signature and $\mathcal{C}$ is a class of Σ-structures, called the *models* of T. Below, let $T = (\Sigma, \mathcal{C})$. A Σ-formula ϕ is *T-satisfiable* if there exists a Σ-structure $\mathcal{M}$ in $\mathcal{C}$ such that ϕ is true in $\mathcal{M}$ under a suitable assignment μ to the free variables of ϕ (in symbols, $\mathcal{M}, \mu \models \phi$); it is *$T$-valid* (in symbols, $T \models \varphi$) if its negation is T-unsatisfiable. Two formulae φ_1 and φ_2 are *T-equivalent* if $\varphi_1 \Leftrightarrow \varphi_2$ is *T-valid.* The *satisfiability modulo the theory T ($SMT(T)$) problem* amounts to establishing the T-satisfiability of quantifier-free Σ-formulae.

A Σ-theory T is *locally finite* if Σ is finite and, for every set $\underline{a}$ of constants, there are finitely many ground terms $t_1, ..., t_{k_{\underline{a}}} \in (\Sigma \cup \underline{a})$, called *representatives*, such that for every ground $(\Sigma \cup \underline{a})$-term u, we have $T \models u = t_i$ for some i. If the representatives are effectively computable from $\underline{a}$ and t_i is computable from u, then T is *effectively* locally finite. For simplicity, we will often say "locally finite" to mean "effectively locally finite". For instance, the pure theory of equality with no function symbols is locally finite. Another example is the *theory of an enumerated data-type* whose signature contains a single sort symbol S with only n constant symbols of sort S and its class of models is such that the interpretation of S is a finite set D of cardinality n and the constants are interpreted as distinct elements of D.

A *T-partition* is a finite set $C_1(\underline{x}), \ldots, C_n(\underline{x})$ of quantifier-free formulae (with free variables contained in the tuple $\underline{x}$) such that $T \models \forall \underline{x} \bigvee_{i=1}^{n} C_i(\underline{x})$ and $T \models \bigwedge_{i \neq j} \forall \underline{x} \neg (C_i(\underline{x}) \wedge C_j(\underline{x}))$. A *case-definable extension* $T' = (\Sigma', \mathcal{C}')$ of a theory $T = (\Sigma, \mathcal{C})$ is obtained from T by applying (finitely many times) the following procedure: (i) take a T-partition $C_1(\underline{x}), \ldots, C_n(\underline{x})$ together with Σ-terms $o_1(\underline{x}), \ldots, o_n(\underline{x})$; (ii) let Σ' be $\Sigma \cup \{F\}$, where F is a "fresh" function symbol (i.e. $F \notin \Sigma$) whose arity is equal to the length of $\underline{x}$; (iii) take as $\mathcal{C}'$ the class of Σ'-structures $\mathcal{M}$ whose restriction to Σ is a model of T and such that

$$\mathcal{M} \models \bigwedge_{i=1}^{n} \forall \underline{x} \, (C_i(\underline{x}) \Rightarrow F(\underline{x}) = o_i(\underline{x})) \, .$$

Thus a case-definable extension T' of a theory T contains finitely many additional function symbols, called *case-defined functions.* It is not hard to effectively translate any $SMT(T')$ problem into an equivalent $SMT(T)$-problem, see [12] for details. In the following, by abuse of notation, we shall identify a theory T and its case-definable extensions T'.

Orderings. A *pre-order* $(P, \leq)$ is the set P endowed with a reflexive and transitive relation. A pre-order $(P, \leq)$ is a *well-quasi-ordering (wqo)* if it is well-founded (i.e. there is no infinite sequence $p_0, p_1, ...$ of elements of P such that $p_{n+1} \leq p_n$) and there is no infinite sequence $p_0, p_1, ...$ of pairwise incomparable elements (i.e. $p_i \not\leq p_j$ for all $i < j$). For example, $\mathbb{N}$ with the usual "less-than-or-equal" relation is a wqo while $\mathbb{Z}$ with the same relation is not.

An *upward closed set* U of the pre-order $(P, \leq)$ is such that $U \subseteq P$ and if $p \in U$ and $p \leq q$ then $q \in U$. A *cone* is an upward closed set of the form $\uparrow p = \{q \in P \mid p \leq q\}$. An upward closed set U is *finitely generated* iff it is a finite union of cones.

Property 1 ([12]). Let $(P, \leq)$ be a wqo. Every upward closed subset of P is finitely generated.

Let P_i be a set of elements and $\leq_i \subseteq P_i \times P_i$ for $i = 1, 2$. Consider the Cartesian product $P_1 \times P_2$ and the following relation on pairs: $(p_1, p_2) \leq (p'_1, p'_2)$ iff $p_1 \leq_1 p'_1$ and $p_2 \leq_2 p'_2$.

Property 2 ([10]). If $(P_1, \leq_1)$ and $(P_2, \leq_2)$ are wqos, then $(P_1 \times P_2, \leq)$ is a wqo.

This can be extended to tuples of $n \geq 2$ elements by a standard inductive argument, i.e. it is possible to show that if $(P_i, \leq_i)$ is a wqo for $i = 1, .., n$, then $(P_1 \times \cdots P_n, \leq)$ is also a wqo where $\langle p_1, ..., p_n \rangle \leq \langle p'_1, ..., p'_n \rangle$ iff $p_i \leq_i p'_i$ for each $i = 1, ..., n$. The property can be used to prove Dickson's lemma stating that that every set of n-tuples of natural numbers has finitely many minimal (with respect to the usual "less-than-or-equal" relation) elements; see again [10] for details.

3 Composed Array-Based Systems

The theory $A^{\langle E_1,...,E_n \rangle}_{\langle I_1,...,I_n \rangle}$ specifies the array data structure manipulated by the class of transition systems considered in the paper. It is parametric with respect to the indexes and elements stored in the arrays, whose algebraic structures are again specified as theories T_{I_k} and T_{E_k}, respectively, for $k = 1, ..., n$. We assume $T_{I_k} = (\Sigma_{I_k}, \mathcal{C}_{I_k})$ to have only one sort symbol $\mathtt{INDEX}_k$. The sorts of the theory $T_{E_k} = (\Sigma_{E_k}, \mathcal{C}_{E_k})$ are given names $\mathtt{ELEM}_{k,\ell}$, where ℓ varies in a given finite index set. We define the composed theory $A^{\langle E_1,...,E_n \rangle}_{\langle I_1,...,I_n \rangle} = (\Sigma, \mathcal{C})$ of arrays with indexes in $T_{I_1}, ..., T_{I_n}$ and elements in $T_{E_1}, ..., T_{E_n}$ as follows. The signature of $A^{\langle E_1,...,E_n \rangle}_{\langle I_1,...,I_n \rangle}$ contains the sort symbols of $T_{I_1}, ..., T_{I_n}, T_{E_1}, ..., T_{E_n}$, together with a new sort symbol $\mathtt{ARRAY}_{k,\ell}$ for each $\mathtt{ELEM}_{k,\ell}$ of Σ_{E_k}, and all the function and predicate symbols in $\Sigma_{I_k} \cup \Sigma_{E_k}$ together with a new function symbol $\_[\_]_{k,\ell} : \mathtt{ARRAY}_{k,\ell}, \mathtt{INDEX}_k \longrightarrow \mathtt{ELEM}^k_\ell$ for each $\mathtt{ELEM}_{k,\ell}$ of Σ_{E_k}. Intuitively, $a[i]_{k,\ell}$

denotes the element of sort $\mathtt{ELEM}_{k,\ell}$ stored in the array a of sort $\mathtt{ARRAY}_{k,\ell}$ at index i; when the sort $\mathtt{ELEM}_{k,\ell}$ is clear from the context, we simply write $a[i]$. The class $\mathcal{C}$ of models of $A^{\langle E_1,\ldots,E_n\rangle}_{\langle I_1,\ldots,I_n\rangle}$ contains a multi-sorted structure $\mathcal{M}$ iff for each sort $\mathtt{ELEM}_{k,\ell}$ of Σ_{E_k}, we have that $\mathtt{ARRAY}^{\mathcal{M}}_{k,\ell}$ is interpreted as the set of (total) functions from $\mathtt{INDEX}^{\mathcal{M}}_k$ to $\mathtt{ELEM}^{\mathcal{M}}_{k,\ell}$, the function symbol $\_[\_]$ is interpreted as function application, and $\mathcal{M}_{|\Sigma_{I_k}}, \mathcal{M}_{|\Sigma_{E_k}}$ are models of T_{I_k} and T_{E_k}, respectively.

A *composed array-based (transition) system (for* $(\langle I_1,...,I_n\rangle, \langle E_1,...,E_n\rangle)$*)* is a triple $\mathcal{S} = (\langle a_1,...,a_n\rangle, \langle I_1,...,I_n\rangle, \langle \tau_1,...,\tau_n\rangle)$ where (i) $a_k = a_k^1, \ldots, a_k^{s_k}$ is a tuple of the array *state variables* (these arrays encode local data of sorts $\mathtt{ELEM}_{k,1}, \ldots, \mathtt{ELEM}_{k,s_k}$, respectively); (ii) $I(a_1,...,a_n)$ is the *initial* formula; and (iii) $\tau(a_1,...,a_n,a'_1,...,a'_n)$ is the *transition* formula, where the prime operator $\cdot'$ uniquely renames the state variables in a tuple of arrays. When $n = 1$, the notion of composed array-based system reduce to that of array-based system [12].

Given a composed array-based system $\mathcal{S} = (\langle a_1,...,a_n\rangle, I, \tau)$ and a formula $U(a_1,...,a_n)$, called the *unsafe* or *goal* formula, (an instance of) the *safety problem* is to establish whether there exists a natural number n such that the formula

$$I(a_1^0,...,a_n^0) \wedge \bigwedge_{k=0}^{n-1} \tau(a_1^k,...,a_n^k,a_1^{k+1},...,a_n^{k+1}) \wedge U(a_1^n,...,a_n^n) \qquad (1)$$

is $A^{\langle E_1,\ldots,E_n\rangle}_{\langle I_1,\ldots,I_n\rangle}$-satisfiable, where a^0 stands for a and a^j for the result of applying $j > 0$ times the prime operator $'$ to a. If there is no such n, then $\mathcal{S}$ is *safe* (w.r.t. U); otherwise, it is *unsafe* and there exists a run (i.e. a sequence of transitions) of length n leading the system from a state in I to a state in U.

3.1 An Example of Composed Array-Based Systems

We consider the security-aware workflow in Figure 1, adapted from [6], and show how it can be represented as a composed array-based system. Figure 1 shows a Petri net [16] that represents the workflow comprising five tasks and their causal dependencies. For instance, task $t1$ becomes enabled when a token is in place $p0$. The result of executing $t1$ is to delete the token in $p0$ and put a token in $p1, p2$, and $p3$. This enables tasks $t2, t3$, and $t4$, respectively, and so on. When the last task $t5$ is executed, resulting in a token in $p7$ while deleting the three tokens in $p4$, $p5$, and $p6$, all tasks in the workflow have been executed. For simplicity, we consider just one workflow instance.

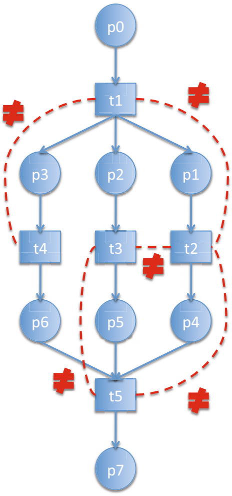


Fig. 1. An example of a constrained workflow

SoD authorization constraints are shown in Figure 1 by means of dashed lines connecting tasks and labelled by $\neq$, meaning that distinct users are required for executing tasks $t1$ and $t2$, $t1$ and $t4$, and so on. BoD constraints can be formalised similarly, using equality instead of inequality labels. Authorization constraints specifying if a user can

Table 1. The constrained workflow system in Figure 1 as a 2-components array-based system $(\langle a_1, a_2\rangle, I(a_1, a_2), \tau(a_1, a_2, a_1', a_2'))$

$a_1 := p0, ..., p7, d\_t1, ..., d\_t5$			$a_2 := a\_t1, ..., a\_t5, t1\_by, ..., t5\_by$	
$I^1(a_1)$	$\forall x.$	$\begin{bmatrix} p0[x] \wedge \neg p1[x] \wedge \neg p2[x] \wedge \\ \neg p3[x] \wedge \neg p4[x] \wedge \neg p5[x] \wedge \\ \neg p6[x] \wedge \neg p7[x] \wedge \neg d\_t1[x] \wedge \\ \neg d\_t2[x] \wedge \neg d\_t3[x] \wedge \\ \neg d\_t4[x] \wedge \neg d\_t5[x] \end{bmatrix}$	$I^2(a_2)$	$\forall u.(\neg t1\_by[u] \wedge \cdots \wedge \neg t5\_by[u])$
$\tau_1^1(a_1, a_1')$	$\exists x.$	$\begin{bmatrix} p0[x] \wedge \neg d\_t1[x] \wedge \\ p0' = \underline{\mathit{false}} \wedge p1' = \underline{\mathit{true}} \wedge \\ p2' = \underline{\mathit{true}} \wedge p3' = \underline{\mathit{true}} \wedge \\ d\_t1' = \underline{\mathit{true}} \end{bmatrix}$	$\tau_1^2(a_2, a_2')$	$\exists u.$ $\begin{bmatrix} a\_t1[u] \wedge \\ t1\_by' = \mathit{upd}(t1\_by, u, \mathit{true}) \end{bmatrix}$
$\tau_2^1(a_1, a_1')$	$\exists x.$	$\begin{bmatrix} p1[x] \wedge \neg d\_t2[x] \wedge \\ p1' = \underline{\mathit{false}} \wedge p4' = \underline{\mathit{true}} \wedge \\ d\_t2' = \underline{\mathit{true}} \end{bmatrix}$	$\tau_2^2(a_2, a_2')$	$\exists u.$ $\begin{bmatrix} a\_t2[u] \wedge \\ \neg t1\_by[u] \wedge \neg t3\_by[u] \wedge \\ t2\_by' = \mathit{upd}(t2\_by, u, \mathit{true}) \end{bmatrix}$
$\tau_3^1(a_1, a_1')$	$\exists x.$	$\begin{bmatrix} p2[x] \wedge \neg d\_t3[x] \wedge \\ p2' = \underline{\mathit{false}} \wedge p5' = \underline{\mathit{true}} \wedge \\ d\_t3' = \underline{\mathit{true}} \end{bmatrix}$	$\tau_3^2(a_2, a_2')$	$\exists u.$ $\begin{bmatrix} a\_t3[u] \wedge \\ \neg t2\_by[u] \wedge \\ t3\_by' = \mathit{upd}(t3\_by, u, \mathit{true}) \end{bmatrix}$
$\tau_4^1(a_1, a_1')$	$\exists x.$	$\begin{bmatrix} p3[x] \wedge \neg d\_t4[x] \wedge \\ p3' = \underline{\mathit{false}} \wedge p6' = \underline{\mathit{true}} \wedge \\ d\_t4' = \underline{\mathit{true}} \end{bmatrix}$	$\tau_4^2(a_2, a_2')$	$\exists u.$ $\begin{bmatrix} a\_t4[u] \wedge \\ \neg t1\_by[u] \wedge \\ t4\_by' = \mathit{upd}(t4\_by, u, \mathit{true}) \end{bmatrix}$
$\tau_5^1(a_1, a_1')$	$\exists x.$	$\begin{bmatrix} p4[x] \wedge p5[x] \wedge p6[x] \wedge \\ \neg d\_t5[x] \wedge \\ p4' = \underline{\mathit{false}} \wedge p5' = \underline{\mathit{false}} \wedge \\ p6' = \underline{\mathit{false}} \wedge p7' = \underline{\mathit{true}} \wedge \\ d\_t5' = \underline{\mathit{true}} \end{bmatrix}$	$\tau_5^2(a_2, a_2')$	$\exists u.$ $\begin{bmatrix} a\_t5[u] \wedge \\ \neg t2\_by[u] \wedge \neg t3\_by[u] \wedge \\ t5\_by' = \mathit{upd}(t5\_by, u, \mathit{true}) \end{bmatrix}$

Goal formula U

$U^1(a_1)$	$\exists x.$	$\begin{bmatrix} \neg p0[x] \wedge \neg p1[x] \wedge \neg p2[x] \wedge \\ \neg p3[x] \wedge \neg p4[x] \wedge \neg p5[x] \wedge \\ \neg p6[x] \wedge p7[x] \wedge \\ d\_t1[x] \wedge d\_t2[x] \wedge d\_t3[x] \wedge \\ d\_t4[x] \wedge d\_t5[x] \end{bmatrix}$	$U^2(a_2)$	*true*

execute a certain task are not shown in Figure 1. We assume that there exist sets *a_t1*, ..., *a_t5* such that $u \in$ *a_t1* iff u is entitled to execute $t1$ and similarly for *a_t2*, ..., *a_t5*.

The composed array-based system $\langle(a_1, a_2), I, \tau\rangle$ defined in Table 1 formalizes the security-aware workflow system in Figure 1. All state variables in a_1, a_2 are Boolean valued[1] arrays, the sort of indexes for array variables in a_1 is S_{I_1}, and the sort of indexes for array variables in a_2 is S_{I_2}. In the rest of this section, we assume that x is a variable of sort I_1 and u is a variable of sort I_2. The theory T_{I_1} (T_{I_2}) is the theory of equality over the sort I_1 (I_2, respectively). Since we consider only one workflow instance, variables in a_1 store the same value for

[1] Booleans are formalized by an enumerated data-type theory with two distinct elements *true* and *false*. For a Boolean valued array a, we abbreviate $a[i] = \mathit{true}$ and $a[i] = \mathit{false}$ as $a[i]$ and $\neg a[i]$, respectively.

every index; intuitively, $p0, ..., p7$ models the presence or absence of a token in the place with the same name in the Petri net of Figure 1 and $d\_t1, ..., d\_t5$, record if tasks $t1, ..., t5$ have been executed or not. Variables in a_2 are "real" arrays indexed over users and $a\_t1[u]$ holds when user u is entitled to execute $t1$ (similarly for $a\_t2, ..., a\_t5$) and $t1\_by[u]$ records the fact that task $t1$ has been executed by user u (similarly for $t2\_by, ..., t5\_by$).

The initial state formula I is the conjunction of I^1 and I^2 that are defined in Table 1. The universal formula $I^1(a_1)$ characterizes the situation in which there is just one token in place $p0$, places $p1, ..., p7$ are empty, and no task has yet been executed. The universal formula $I^2(a_2)$ says that no user has yet executed any task. Notice how I^1 contains just the states variables in a_1 concerning the Petri net and I^2 only the state variables in a_2 concerning the authorization constraints (as $a\_t1, ..., a\_t5$ do not occur in I^2, they are left unconstrained).

The transition formula τ is the disjunction of $\tau^1_k \wedge \tau^2_k$ where τ^1_k and τ^2_k are shown in Table 1 ($k = 1, ..., 5$). For the sake of compactness, we have used the following abbreviations in writing the formulae: $\underline{true}$ ($\underline{false}$) is the function returning *true* (*false*, respectively) for any input and $upd(t\_by, u, true)$ is the function that returns the same value of t_by for every user except in u for which it returns *true* ($t \in \{t1, ..., t5\}$). Array variables not occurring in $\tau^1_k \wedge \tau^2_k$ are updated identically, i.e. the formula $\tau^1_k \wedge \tau^2_k$ abbreviates $\tau^1_k \wedge \tau^2_k \wedge \bigwedge_{a \in A} a' = a$ where A contains all those state variables not mentioned in $\tau^1_k \wedge \tau^2_k$. For instance, τ^1_1 formalizes the enabled condition (there is a token in $p0$ and $t1$ has not yet been executed) and the effect (delete the token in $p0$, put a token in $p1, p2$, and $p3$, and set to true the fact that $t1$ has been executed) corresponding to the execution of task $t1$ in the Petri net of Figure 1. Instead, τ^2_1 formalizes the authorization condition for executing task $t1$: a user u should be entitled to execute $t1$ and record the fact that u has executed $t1$. More interestingly, τ^2_2 besides requiring the user u to be entitled to execute $t2$, it also requires that u is not the same user that has executed both $t1$ and $t3$: this corresponds to the two SoD constraints represented in Figure 1 as the two dashed lines between $t1$ and $t2$ and between $t2$ and $t3$. The intuitive reading of the remaining formulae can be derived in a similar way.

Two observations are important. First, the variables in a_1 are always updated in such a way to store the same value at all indexes. Second, the variables $a\_t1, ..., a\_t5$ in a_2 are unchanged by τ, i.e. the capability of users to execute tasks does not change over time. This is not always the case; for instance, users can delegate permissions to execute certain tasks to other users during workflow execution (for more on this point, see Section 6 below).

A first sanity check of the design of the security-aware workflow in Figure 1 is to verify if the situation in which there is just one token in $p7$ and all tasks have been executed can be reached. Formally, this amounts to solve the safety problem involving the composed array-based system $\langle (a_1, a_2), I, \tau \rangle$ and the goal formula U obtained by conjoining the two existential formulae $U^1(a_1)$ and $U^2(a_2)$ shown in Table 1. As it was the case for I^1 and I^2 above, U^1 contains only the state variables in a_1 and U^2 only those in a_2. In this particular case, U^2 holds for any value of the variables in a_2, thereby leaving them unconstrained.

3.2 The Class of n-Components Array-Based Systems

By generalizing the example in Section 3.1, we introduce a sub-class of composed array-based systems. Preliminary, we need to introduce the following notational conventions (adopted from [1]). An underlined variable name abbreviates a tuple of variables of unspecified (but finite) length and, if $\underline{i} := i_1, \ldots, i_n$, the notation $a[\underline{i}]$ abbreviates the $s*n$-tuple of terms $a_1[i_1], \ldots, a_1[i_n], \ldots, a_s[i_1], \ldots, a_s[i_n]$. To simplify notation, we underline symbols $\underline{i}, \underline{e}, \ldots$ for tuples of elements and index variables, whereas we use just a (not underlined) for the tuple $a_1, \ldots, a_s$ of array variables. Possibly sub-/super-scripted expressions of the form $\phi(\underline{i}, \underline{e}), \psi(\underline{i}, \underline{e})$ denote *quantifier-free* $(\Sigma_I \cup \Sigma_E)$*-formulae* in which at most the variables $\underline{i} \cup \underline{e}$ occur. Also, $\phi(\underline{i}, \underline{t}/\underline{e})$ (or simply $\phi(\underline{i}, \underline{t})$) abbreviates the substitution of the Σ-terms $\underline{t}$ for the variables $\underline{e}$. Thus, for instance, $\phi(\underline{i}, a[\underline{i}])$ denotes the formula obtained by replacing $\underline{e}$ with $a[\underline{i}]$ in the quantifier-free formula $\phi(\underline{i}, \underline{e})$. An *$a$-$\forall^I$-formula* is a formula of the form $\forall \underline{i}. \phi(\underline{i}, a[\underline{i}])$. An *$a$-$\exists^I$-formula* is a formula of the form $\exists \underline{i}. \phi(\underline{i}, a[\underline{i}])$. An *$a$-$\exists^I\forall^I$-formula* is a formula of the form $\exists \underline{i}\ \forall \underline{j}\ \psi(\underline{i}, \underline{j}, a[\underline{i}], a[\underline{j}])$.

A *n-components array-based system* $\langle (a_1, ..., a_n), I, \tau \rangle$ is a composed array-based system where a_t is a tuple of array variables of sort $\mathtt{ARRAY}_{t,\ell}$,

$$\begin{aligned} I(a_1, ..., a_n) &:= I^1(a_1) \wedge \cdots \wedge I^n(a_n) \\ \tau(a_1, ..., a_n, a_1', ..., a_n') &:= \bigvee_{k \in K} \left(\tau_k^1(a_1, a_1') \wedge \cdots \wedge \tau_k^n(a_1, a_1') \right), \end{aligned}$$

K is a finite set, $I^t(a_t)$ is an a_t-$\forall^I$-formula, I is called an *n-components initial formula*, and $\tau_k^t(a_t, a_t')$ is in *functional form*, i.e. a formula of the form

$$\exists \underline{i} \left(\phi_L(\underline{i}, a_t[\underline{i}]) \wedge \forall j. a_t'[j] = F(\underline{i}, a_t[\underline{i}], j, a_t[j]) \right), \qquad (2)$$

the quantifier-free formula ϕ_L is the *guard* and $F = F_1, \ldots, F_s$ is a tuple of case-defined functions, called the *updates* ($t = 1, ..., n$). An *n-components unsafe or goal formula U* is of the form $U^1(a_1) \wedge \cdots \wedge U^n(a_n)$ for U^t an a_t-$\exists^I$-formula ($t = 1, ..., n$).

It is easy to see that the composed array-based system of Section 3.1 is a 2-components array-based systems.

4 Backward Reachability

A general approach to solve instances of the safety problem is based on the symbolic computation of the set of backward reachable states. For $b \geq 0$, the *b-pre-image* of a n-components goal formula $H(a_1, ..., a_n)$ is $Pre^0(\tau, H) := H$ and $Pre^{b+1}(\tau, H) := Pre(\tau, Pre^b(\tau, H))$, where

$$Pre(\tau, H) := \exists a_1', ..., a_n'. (\tau(a_1, ..., a_n, a_1', ..., a_n') \wedge H(a_1', ..., a_n')). \qquad (3)$$

Given an n-components array-based system $\langle (a_1,...,a_n), I, \tau\rangle$ and an n-components goal formula $U(a_1,...,a_n)$, the formula $Pre^b(\tau, U)$ describes the set of backward reachable states in b steps (for $b \geq 0$).

The procedure to establish if the n-components goal formula U is reachable is based on iteratively computing the symbolic representations of the set $BR(a_1,...,a_n)$ of states from which it is possible to reach U, by applying—finitely many times—the transition τ. Formally, we define $BR^b(\tau, U)$ to be the disjunction of $Pre^i(\tau, U)$ for $i = 0,...,b$ with $b \geq 0$. $BR^b(\tau, U)$ represents the set of states which are backward reachable from the states in U in at most b steps. In order to stop computing formulae in the sequence $BR^b(\tau, U)$, there are two criteria. **(C1)** check whether $BR^b(\tau, U) \wedge I$ is $A^{\langle E_1,...,E_n\rangle}_{\langle I_1,...,I_n\rangle}$-satisfiable: in this case, there exists a finite sequence of transitions in τ that leads the system from an initial state in I to a state in U. **(C2)** check whether $BR^{b+1}(\tau, U) \Rightarrow BR^b(\tau, U)$ is $A^{\langle E_1,...,E_n\rangle}_{\langle I_1,...,I_n\rangle}$-valid or, by refutation, if $BR^{b+1}(\tau, U) \wedge \neg BR^b(\tau, U)$ is $A^{\langle E_1,...,E_n\rangle}_{\langle I_1,...,I_n\rangle}$-unsatisfiable: in this case, BR^b is the *fix-point* of the sequence of BR^i's.

To ensure that formulae to be checked for criteria **(C1)** and **(C2)** have the same shape at each iteration, the class of formulae used to represent goal states must be closed under pre-image computation.

> From now on, we fix a n-components array-based system $\mathcal{S} = \langle (a_1,...,a_n), I, \tau\rangle$ where $\tau = \bigvee_{k\in K} \bigwedge_{t=1}^n \tau_k^t(a_t, a'_t)$ and τ_k^t is in functional form (2) together with an n-components goal formula $H(a_1,...,a_n)$ of the form $\bigwedge_{t=1}^n H^t(a_t)$ with H^t an a_t-$\exists^I$-formula ($t = 1,...,n$).

Proposition 1. *The pre-image $Pre(\tau, H)$ of H with respect to τ is logically equivalent to*

$$\bigvee_{k\in K} \bigwedge_{t=1}^{n} Pre(\tau_k^t, H^t)\,, \tag{4}$$

where $Pre(\tau_k^t, H^t)$ is logically equivalent to an (effectively computable) a_t-$\exists^I$-formula for $t = 1,...,n$.

The proof consists of simple logical manipulations. An important consequence of this property is the possibility of manipulating each component formula separately and then form the overall pre-image by Boolean combination. Another consequence is that the symbolic representation $BR^b(\tau, U)$ of the set of backward reachable states is logically equivalent to (an effectively computable) disjunction of n-components goal formulae for $b \geq 0$, i.e. a formula of the form

$$\bigvee_{j\in J} \bigwedge_{t=1}^{n} H_j^t(a_t) \tag{5}$$

for J a finite set and H_j^t an a_t-$\exists^I$-formula for $j \in J$ and $t = 1,...,n$. For efficiency, it is important to delete unsatisfiable disjuncts in (5) that result from the fact that a disjunct of the transition is not applicable. We give sufficient conditions for the decidability of the satisfiability of a-$\exists^I\forall^I$-formulae that, in turn, implies the decidability of the satisfiability of any disjunct in (5) since a_t-$\exists^I$-formulae are also a_t-$\exists^I\forall^I$-formulae for $t = 1, ..., n$.

Proposition 2. *Assume that (**TH1**) the SMT(T_{I_t}) and SMT(T_{E_t}) problems are decidable, and (**TH2**) T_{I_t} is locally finite and its class of models is closed under sub-structures, for $t = 1, ..., n$. Furthermore, let AE be a formula of the form $\bigwedge_{t=1}^{n} AE^t(a_t)$ with $AE^t(a_t)$ a $\exists^I\forall^I$-sentence for $t = 1, ..., n$. Then*

1. *AE is $A_{\langle I_1,...,I_n\rangle}^{\langle E_1,...,E_n\rangle}$-satisfiable iff $AE^t(a_t)$ is $A_{I_t}^{E_t}$-satisfiable for each $t = 1, ..., n$,*
2. *the $A_{I_t}^{E_t}$-satisfiability of $AE^t(a_t)$ is decidable for any $t \in \{1, ..., n\}$.*

After showing closure under pre-image computation, we must ensure that criteria **(C1)** and **(C2)** above are decidable.

> From now on, without loss of generality, we assume that $BR^b(\tau, U)$ stands for a formula of the form (5) in which all disjuncts are $A_{\langle I_1,...,I_n\rangle}^{\langle E_1,...,E_n\rangle}$-satisfiable.

Proposition 3. *Assume (**TH1**) and (**TH2**) as in Proposition 2. Then*

1. *$BR^b(\tau, H)$ is logically equivalent to a formula of the form $\bigvee_{j\in J}\bigwedge_{t=1}^{n} H_j^t(a_t)$ with $H_j^t(a_t)$ an $A_{I_t}^{E_t}$-satisfiable a_t-$\exists^I$-formula for $j \in J$ and $t = 1, ..., n$,*
2. *$BR^b(\tau, H) \wedge I$ is $A_{\langle I_1,...,I_n\rangle}^{\langle E_1,...,E_n\rangle}$-satisfiable iff there exists $j \in J$ such that $H_j^t(a_t) \wedge I^t(a_t)$ is $A_{I_t}^{E_t}$-satisfiable for each $t = 1, ..., n$.*
3. *$BR^{b+1}(\tau, H) \wedge \neg BR^b(\tau, H)$ is $A_{\langle I_1,...,I_n\rangle}^{\langle E_1,...,E_n\rangle}$-unsatisfiable iff for each disjunct $\bigwedge_{t=1}^{n} L^t(a_t)$ of $BR^{b+1}(\tau, H)$, we have that $L^t(a_t) \wedge \bigwedge_{j\in J} \neg H_j^t(a_t)$ is $A_{I_t}^{E_t}$-unsatisfiable for each $t = 1, ..., n$, where $L^t(a_t)$ is an $A_{I_t}^{E_t}$-satisfiable a_t-$\exists^I$-formula for $t = 1, ..., n$, and*
4. *the $A_{I_t}^{E_t}$-satisfiability of both $H_j^t(a_t) \wedge I^t(a_t)$ and $L^t(a_t) \wedge \bigwedge_{j\in J} \neg H_j^t(a_t)$ is decidable ($t = 1, ..., n$).*

This proposition allows us to reduce criteria (**C1**) and (**C2**) above to finitely many satisfiability checks involving only component formulae.

Figure 2 presents the backward reachability algorithm as the function BReach based on the propositions above. First, we briefly describe the auxiliary functions *Pre*, simplify, emptyint?, and entail?. The fact that *Pre* computes disjunctions of n-components goal formulae is guaranteed by Proposition 1. The function simplify preserves this by taking a formula of the form $\bigvee_{j\in J}\bigwedge_{t=1}^{n} H_j^t(a_t)$ and returning $\bigvee_{j\in J'}\bigwedge_{t=1}^{n} H_j^t(a_t)$ such that $J' \subseteq J$ with with $H_j^t(a_t)$ an $A_{I_t}^{E_t}$-satisfiable a_t-$\exists^I$-formula, for each $j \in J'$ and $t = 1, ..., n$. The function emptyint? returns true iff the conjunction of the initial formula I and a formula of the form (5) representing (an approximation of) the set of backward reachable states is $A_{\langle I_1,...,I_n\rangle}^{\langle E_1,...,E_n\rangle}$-unsatisfiable by using point 2 of Proposition 3. The function entail? returns true iff the formula of the form (5) passed as the first argument implies the formula of the same form passed as the second modulo $A_{\langle I_1,...,I_n\rangle}^{\langle E_1,...,E_n\rangle}$ by using

```
function BReach($\mathcal{S}$, $U$)
1  $P \longleftarrow U$; $B \longleftarrow \bot$;
2  while not entail?$(P, B)$ do
3      if not emptyint?$(I, P)$ then return unsafe;
4      $B \longleftarrow P \vee B$;
5      $P \longleftarrow$ simplify$(Pre(\tau, P))$;
6  end
7  return (safe, $B$);
```

Assuming $\mathcal{S} = \langle (a_1, ..., a_n), I, \tau \rangle$ is an n-components array-based system where

$$I(a_1, ..., a_n) := \bigwedge_{t=1}^{n} I^t(a_t) \quad \text{and} \quad \tau(a_1, ..., a_n, a'_1, ..., a'_n) := \bigvee_{k \in K} \bigwedge_{t=1}^{n} \tau^t_k(a_t, a'_t)$$

with I^t an a_t-$\forall^I$-formula, $\tau^t_k(a_t, a'_t)$ a transition formula in functional form (2), and $U(a_1, ..., a_n)$ is an n-components goal formula of the form $\bigwedge_t U^t(a_t)$ in which U^t is an a_t-$\exists^I$-formula (for $t = 1, ..., n$).
Conditions (**TH1**) and (**TH2**) of Proposition 2 on T_{I_t} and T_{E_t} hold for $t = 1, ..., n$.

Fig. 2. Symbolic backward reachability for n-components array-based systems

point 3 of Proposition 3. By point 4 of the same proposition, we derive the decidability of the satisfiability checks—under assumptions (**TH1**) and (**TH2**)—in emptyint? and entail? and thus also their effectiveness. In turn, this ensures the effectiveness of BReach, that can be described as follows.

At the b-th iteration of the loop, BReach stores in the variable B the formula $BR^b(\tau, U)$ representing the set of states which are backward reachable from the states in U in at most b steps (whereas the variable P stores the formula $Pre^b(\tau, U)$). While computing $BR^b(\tau, U)$, BReach also checks whether (line 3) the system is unsafe by invoking empty? on I and $Pre^b(\tau, U)$, or (line 2) a fix-point has been reached by by invoking entail? on $BR^b(\tau, U)$ and $BR^{b-1}(\tau, U)$. When BReach returns the safety of the system (line 7), the variable B stores the formula describing the set of states which are backward reachable from U which is also a fix-point.

5 Termination of Backward Reachability

As observed in Section 3, 1-component array-based systems are the same as the array-based systems of [12]. The undecidability of safety problems for the latter—even when assumptions (**TH1**) and (**TH2**) of Proposition 2 hold (see [12] for a proof of this by a reduction to the halting problem of a Minsky machine)—implies that BReach may non terminate. Fortunately, it is possible to identify sufficient conditions to guarantee the decidability of the safety problem that are also useful in applications. The idea is to introduce a suitable model-theoretic notion of *configurations* to be the semantic counter-part of n-components goal formulae, and then show that a wqo (recall the definition in Section 2) can be defined on them, which implies the termination of BReach (see Theorem 1 below) and thus the decidability of the safety problem. The partial ordering $\preceq$

on configurations is defined by modularly reusing those $\preceq_1, ..., \preceq_n$ defined on the n-components. If each $\preceq_t$ is a wqo, then also $\preceq$ is so by Property 2 (in Section 2).

A *state* of an n-components array-based systems $\mathcal{S}$ is a tuple $\langle(s_1, \mathcal{M}_1), ..., (s_n, \mathcal{M}_n)\rangle$ where $\mathcal{M}_t$ is a model of $A_{I_t}^{E_t}$ and $s_t \in \mathtt{ARRAY}_{t,\ell}^{\mathcal{M}_t}$ for $t = 1, ..., n$. The *t-component of a state* $\langle(s_1, \mathcal{M}_1), ..., (s_n, \mathcal{M}_n)\rangle$ is the pair $(s_t, \mathcal{M}_t)$ for $t \in \{1, ..., n\}$. A *configuration* is a tuple $\langle(s_1, \mathcal{M}_1), ..., (s_n, \mathcal{M}_n)\rangle$ of pairs such that s_t is an array of a finite index model $\mathcal{M}_t$ (i.e. a structure in which the interpretation of the sort of index is a finite set) of $A_{I_t}^{E_t}$; $\mathcal{M}_t$ is omitted whenever it is clear from the context ($t = 1, ..., n$). Notice that the set of configurations is a sub-set of the set of states. We associate a Σ_{I_t}-structure s_{I_t} and a Σ_{E_t}-structure s_{E_t} with the t-component $(s_t, \mathcal{M}_t)$ of an $A_{\langle I_1,...,I_n\rangle}^{\langle E_1,...,E_n\rangle}$-configuration as follows: the Σ_{I_t}-structure s_{I_t} is simply the finite structure $\mathcal{M}_{I_t}$ whereas s_{E_t} is the smallest Σ_{E_t}-substructure of $\mathcal{M}_{E_t}$ containing the image of s_t. Intuitively, a configuration is a finite representation of a possibly infinite set of states that "contains at least the part mentioned in the configuration." This can be formalized by defining a pre-order $\preceq$ over configurations as follows. Preliminary, recall that an embedding is a homomorphism that preserves and reflects relations and operations (see, e.g., [14] for a formal definition).

For each t-component of a configuration ($t = 1, ..., n$), define a pre-order $\preceq_t$ as follows: $s_t \preceq s'_t$ iff there exist a Σ_{I_t}-embedding $\mu_t : s'_{I_t} \longrightarrow s_{I_t}$ and a Σ_{E_t}-embedding $\nu_t : s'_{E_t} \longrightarrow s_{E_t}$ such that the set-theoretical compositions of μ_t with s_t and of s'_t with ν_t are equal, for each $t = 1, ..., n$. For every pair of configurations $\langle s_1, ..., s_n\rangle$ and $\langle s'_1, ..., s'_n\rangle$, we say that $\langle s'_1, ..., s'_n\rangle \preceq \langle s_1, ..., s_n\rangle$ iff $s_t \preceq_t s'_t$ for each $t = 1, ..., n$.

Define the set $[[H]]$ of states denoted by the formula H as

$$\{\langle(s_1, \mathcal{M}_1), ..., (s_n, \mathcal{M}_n)\rangle \mid \mathcal{M}_t, s_t \models H_t \text{ for each } t = 1, ..., n\}.$$

Theorem 1. $\mathsf{BReach}(\mathcal{S}, H)$ *terminates if (*i*) assumptions (**TH1**) and (**TH2**) hold (as in Proposition 2), (*ii*)* T_{E_t} *is locally finite, and (*iii*) the pre-order* $\preceq_t$ *on the t-component of* $A_{\langle I_1,...,I_n\rangle}^{\langle E_1,...,E_n\rangle}$*-configurations is a wqo (for* $t = 1, ..., n$*).*

Proof. Assumptions (**TH1**) and (**TH2**)—together with the particular shape of the formulae in the n-components array-based system $\mathcal{S}$—are needed for Breach to be a semi-algorithm according to Proposition 3. For termination, we distinguish two cases according to the fact that $\mathcal{S}$ is either safe or unsafe w.r.t. H. When $\mathcal{S}$ is unsafe with respect to H, we know that BReach terminates because it returns at line 3 of Figure 2. We now consider the case in which $\mathcal{S}$ is safe with respect to H. Here, the crux is to show that it is possible to compute an n-components goal formula that represent the set $B(\tau, H)$ of configurations that are backward reachable from the configurations satisfying H. Under the assumption that T_{E_t} is locally finite for each $t = 1, ..., n$, we can show that

(*) the termination of $\mathsf{BReach}(\mathcal{S}, H)$ is equivalent to the fact that the set $B(\tau, H)$ of configurations is a finitely generated upset.

A sufficient condition to guarantee that $B(\tau, H)$ is a finitely generated upset is that the pre-order $\preceq$ on $A_{\langle I_1,...,I_n\rangle}^{\langle E_1,...,E_n\rangle}$-configurations is a wqo. This is guaranteed

Table 2. Wqos on t-components of n-components array-based systems

T_{I_t}	T_{E_t}	$\preceq_t$ is a wqo by
Pure equality	Enumerated data-type	Dickson lemma
Total order	Enumerated data-type	Higman lemma
Pure equality	Rationals with $<$	Kruskal theorem

by the assumption that each pre-order $\preceq_t$ on $A_{I_t}^{E_t}$-configurations is a wqo and by the observation after Property 2 at the end of Section 2.

We are then left with the task of showing (*). For this, we preliminary observe that the following three claims hold:

(Claim 1) for every n-components goal formula H, the set $[[H]]$ is upward closed;

(Claim 2) for every pair H_1, H_2 of n-components goal formulae, we have that $[[H_1]] \subseteq [[H_2]]$ iff $(H_1 \Rightarrow H_2)$ is $A_{\langle I_1,\ldots,I_n\rangle}^{\langle E_1,\ldots,E_n\rangle}$-valid;

(Claim 3) finitely generated upsets of $A_{\langle I_1,\ldots,I_n\rangle}^{\langle E_1,\ldots,E_n\rangle}$-configurations coincide with sets of $A_{\langle I_1,\ldots,I_n\rangle}^{\langle E_1,\ldots,E_n\rangle}$-configurations of the form $[[H]]$, for some n-components goal formula H.

These claims are extensions of similar results in [12] for array-based systems.

We now consider the two sides of the bi-conditional (*). Preliminary, we observe that $B(\tau, H) = \bigcup_{b\geq 0}[[BR^b(\tau, H)]]$ by Claim 3.

$B(\tau, H)$ is finitely generated implies the termination of $\mathsf{BReach}(\mathcal{S}, H)$. $B(\tau, H)$ is an upward closed set since $B(\tau, K)$ is the union of upward closed sets and $[[H]]$ is so by (Claim 1). Because of (Claim 2), we have that

$$[[BR^0(\tau, H)]] \subseteq [[BR^1(\tau, H)]] \subseteq \cdots \subseteq [[BR^b(\tau, H)]] \subseteq [[BR^{b+1}(\tau, H)]] \subseteq \cdots$$

Since $B(\tau, H)$ is finitely generated, there exists b such that $[[BR^b(\tau, H)]] = [[BR^{b+1}(\tau, H)]]$ and, again by (Claim 2), we derive that $BR^b(\tau, H) \Leftrightarrow BR^{b+1}(\tau, H)$ is $A_{\langle I_1,\ldots,I_n\rangle}^{\langle E_1,\ldots,E_n\rangle}$-valid, i.e. BReach terminates.

The termination of $\mathsf{BReach}(\mathcal{S}, H)$ *implies that $B(\tau, H)$ is finitely generated.* The termination of $\mathsf{BReach}(\mathcal{S}, H)$ is equivalent to have the $A_{\langle I_1,\ldots,I_n\rangle}^{\langle E_1,\ldots,E_n\rangle}$-validity of $BR^b(\tau, H) \Leftrightarrow BR^{b+1}(\tau, H)$. This is equivalent to $[[BR^b(\tau, H)]] = [[BR^{b+1}(\tau, H)]]$ by (Claim 2), for some $b \geq 0$. □

The difficulty in applying Theorem 1 is to show that $\preceq_t$ is a wqo. Table 2 provides some help in this respect. It is possible to show that all the theories in the table satisfy assumptions (**TH1**) and (**TH2**) of Proposition 2 and that the T_{E_t}'s are locally finite (see, e.g., [12]). The last column of the table reports that it is possible to prove that the pre-order $\preceq_t$ on the t-component of configurations is a wqo by well-known results (see [10] for a survey). These observations and Theorem 1 implies the following result.

Corollary 1. *If T_{I_t} and T_{E_t} are in Table 2 for each $t = 1, ..., n$, then the safety problem for $\mathcal{S}$ and H is decidable.*

Even in case of termination, the complexity of BReach may be non-primitive recursive; this is inherited from the backward reachability array-based systems in [12]. Since safety and fix-point checks can be reduced to sequences of satisfiability checks of component formulae, most of the heuristics developed for MCMT [13] or Cubicle [5] (two model checkers for array-based systems), can be re-used to implement BReach that can terminate in reasonable time on problems relevant to applications. This is left as future work.

6 Application to Security-Aware Workflow Systems

We can apply Corollary 1 to show the decidability of the safety problem for the 2-components array-based system of Section 3.1. Along the lines of Section 3.2, this can be generalized by defining a *constrained workflow system* as a 2-components array-based system in which T_{I_t} is the pure theory of equality and T_{E_t} is the enumerated data-type theory of the Booleans, for $t = 1, 2$. With this notion of constrained workflow systems, SoD or BoD constraints can be imposed on tasks in different instances of a workflow and not only to tasks in the same instance, called inter-instance and intra-instance constraints, respectively, in [15]. Inter-instance constraints are crucial to limit frauds by, e.g., preventing coalitions among malevolent users. Moreover, we can express delegation, i.e. a user can transfer part of its permissions to execute tasks to another user. In our framework, this is achieved by considering authorizations as state variables. In the constrained workflow system of Section 3.1, we can specify the situation in which user u_1 with the permission to execute $t3$ can delegate this capability to user u_2 capable of executing $t2$ as follows: $\exists u_1, u_2.(a\_t3[u_1] \wedge \neg a\_t3[u_2] \wedge a\_t2[u_2] \wedge a\_t3' = upd(a\_t3, u_2, true))$. By Corollary 1, we can derive the following result.

Theorem 2. *The safety problem for constrained workflow systems is decidable.*

A natural extension of the above notion of constrained workflow systems, advocated in [2], consists of taking into account conditions involving the data processed by the tasks in the workflow. The advantage of adopting this extended model is to reduce the non-determinism introduced by abstracting away the dependencies of the control-flow from data values and, ultimately, to design more precise analysis techniques, i.e. returning fewer spurious error traces. Interestingly, it is possible to accommodate this extension in our framework while retaining the decidability of the safety problem. A *constrained workflow system with numerical data-flow* is a 3-components array-based system where T_{I_t} is the pure theory of equality ($t = 1, 2, 3$), T_{E_t} is the enumerated data-type theory of the Booleans ($t = 1, 2$), and T_{E_3} is the theory of rationals with the standard ordering relation $<$ (recall Table 2). Although abstract, this class of systems allows us to express situations in which a task can only be executed when the value of a numeric variable is, e.g., lower (or greater) than a given threshold. For example, in a bank workflow handling loans, there can be two types of evaluation of the economic situation of a client: short ($t1$) and comprehensive ($t2$). If the requested amount of the loan is below 10,000 Euro then $t1$ is executed; otherwise, $t2$ is performed. By Corollary 1, we can also derive the following result.

Theorem 3. *The safety problem for constrained workflow systems with numerical data-flow is decidable.*

7 Conclusions and Related Work

We have introduced the class of composed array-based systems and studied the decidability of the (parametric) safety problem for the sub-class of n-components array-based systems. For this, we have designed a backward reachability procedure that lift that for array-based systems in [12] by modularly re-using SMT solving. We have applied our results to prove the decidability of the safety problems for two classes of security-aware workflow systems.

Constrained workflow systems have been the subject of a long line of research in security; see, e.g., [7] for very recent work and pointers to the literature. Our notion of constrained workflow system generalizes that in [7] in several respects. We describe systems with a finite (but unknown) number of workflow instances or users, we handle loops (i.e. a certain set of tasks can be repeated a finite but unknown number of times) and delegation. All these are not considered in [7]. As a consequence, the scope of applicability of Theorem 2 is much wider than the corresponding result in [7]. However, [7] focuses on the problem of guaranteeing the termination of the workflow while satisfying authorization constraints (at run-time), called the *workflow satisfiability problem* (*WSP*). In this context, it is shown that the knowledge acquired by solving certain safety problems can be used to simplify the solution of WSPs. An interesting line of future work is to take advantage of the generality of our approach to solve WSPs. In particular, the use of the formula representing the fix-point (of the set of reachable states) returned by BReach may be used to build an algorithm solving the WSP on top of an SMT solver.

The work in [2] describes a model checking technique to check temporal properties of security-aware workflows. Decidability is not discussed and the verification technique considers a single workflow instance and a bounded number of users. In contrast, Theorem 3 guarantees the decidability of the safety problem for constrained workflow systems with numerical data-flow regardless of the number of users or the number of workflow instances. However, [2] allows for the verification of arbitrary temporal properties. As future work, we plan to extend to n-components array-based systems the decidability result in [11] for a class of liveness (progress) properties of array-based systems.

The decidability result in [3] can be seen as an instance of Theorem 3. In fact, [3] considers only a sub-class of the transitions that can be specified with the notion of n-components array-based system introduced in this paper.

The "pid quantified constraints" introduced in [9] can be seen as 1-composed array-based systems. Theorem 1 is an answer to the open problem (stated in [9]) of finding conditions to guarantee the termination of fix-point computations for n-components array-based systems. However, the model checking technique in [9] aims to prove temporal properties expressed in Computation Tree Logic.

With [4], we share the goal of modularly re-using techniques to handle symbolic constraints for infinite state model checking. The main difference is that we exploit SMT solving whereas [4] adopts a combination of Boolean reasoning and a decision procedure for Pressburger Arithmetic.

Acknowledgments. The work of the first author is supported by the RESTATE Programme, co-funded by the European Union under the FP7 COFUND Marie Curie Action—Grant agreement no. 267224, and that of the second author is partially supported by the "Automated Security Analysis of Identity and Access Management Systems (SIAM)" project funded by Provincia Autonoma di Trento in the context of the "team 2009 - Incoming" COFUND action of the European Commission (FP7).

References

1. Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., Rossi, G.P.: Universal Guards, Relativization of Quantifiers, and Failure Models in Model Checking Modulo Theories. J. on Satisfiability, Boolean Modeling and Comp. (JSAT) 8, 29–61 (2012)
2. Armando, A., Ponta, S.E.: Model Checking of Security-Sensitive Business Processes. In: Degano, P., Guttman, J.D. (eds.) FAST 2009. LNCS, vol. 5983, pp. 66–80. Springer, Heidelberg (2010)
3. Armando, A., Ranise, S.: Automated Analysis of Infinite State Workflows with Access Control Policies. In: Meadows, C., Fernandez-Gago, C. (eds.) STM 2011. LNCS, vol. 7170, pp. 157–174. Springer, Heidelberg (2012)
4. Bultan, T., Gerber, R., League, C.: Composite Model Checking: Verification with Type-Specific Symbolic Representations. ACM TOSEM 9(1), 3–50 (2000)
5. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 718–724. Springer, Heidelberg (2012)
6. Crampton, J.: A reference monitor for workflow systems with constrained task execution. In: 10th ACM SACMAT, pp. 38–47. ACM (2005)
7. Crampton, J., Gutin, G.: Constraint expressions and workflow satisfiability. In: 18th ACM SACMAT. ACM (2013)
8. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press, New York (1972)
9. Fu, X., Bultan, T., Su, J.: Formal Verification of e-Services and Workflows. In: Bussler, C.J., McIlraith, S.A., Orlowska, M.E., Pernici, B., Yang, J. (eds.) WES 2002. LNCS, vol. 2512, pp. 188–202. Springer, Heidelberg (2002)
10. Gallier, J.H.: What's So Special About Kruskal's Theorem and the Ordinal Γ_0? A Survey of Some Results in Proof Theory. APAL 53(3), 199–260 (1991)
11. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT Model Checking of Array-Based Systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 67–82. Springer, Heidelberg (2008)
12. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. In: LMCS, vol. 6(4) (2010)
13. Ghilardi, S., Ranise, S.: MCMT: A Model Checker Modulo Theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 22–29. Springer, Heidelberg (2010)
14. Hodges, W.: Model Theory. Cambridge University Press (1993)
15. Warner, J., Atluri, V.: Inter-Instance Authorization Constraints for Secure Workflow Managment. In: SACMAT, pp. 190–199. ACM (2006)
16. Murata, T.: Petri nets: properties, analysis and applications. Proc. of the IEEE 77(4), 541–580 (1989)

Presburger Arithmetic in Memory Access Optimization for Data-Parallel Languages

Ralf Karrenberg[1], Marek Košta[2], and Thomas Sturm[2]

[1] Saarland University, 66123 Saarbrücken, Germany
karrenberg@cs.uni-saarland.de
[2] Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany
{mkosta,sturm}@mpi-inf.mpg.de

Abstract. Data-parallel languages like OpenCL and CUDA are an important means to exploit the computational power of today's computing devices. We consider the compilation of such languages for CPUs with SIMD instruction sets. To generate efficient code, one wants to statically decide whether or not certain memory operations access consecutive addresses. We formalize the notion of consecutivity and algorithmically reduce the static decision to satisfiability problems in Presburger Arithmetic. We introduce a preprocessing technique on these SMT problems, which makes it feasible to apply an off-the-shelf SMT solver. We show that a prototypical OpenCL CPU driver based on our approach generates more efficient code than any other state-of-the-art driver.

1 Introduction

Data-parallel languages like OpenCL and CUDA are ubiquitous in today's computing landscape. They follow the so-called SPMD (Single Program, Multiple Data) paradigm, where the technical details of parallelization are abstracted away: The programmer writes a scalar function, called the *kernel*. The kernel is executed in multiple *work items* (sometimes ambiguously called *threads*) by a runtime system. To make every work item perform an individual task, e.g. writing to different elements of an array, special primitives are built into the language to query the *ID* of a work item.

Due to these semantics, the runtime system may choose to execute work items in parallel. On GPUs, this boils down to scheduling each work item to one of the hardware-managed threads via the device driver. On CPUs, the same scheme can be used by employing well-known libraries like pthreads, OpenMP, or MPI to exploit all available cores. In addition, today's CPUs offer another level of parallelism *per core* in the form of SIMD instructions. These are instructions that perform the same operation on multiple input values at once (Single Instruction, Multiple Data). This saves both execution time and power consumption. Fig. 1 depicts variants of how the runtime system could choose to execute a 2-dimensional grid of work items.

The historical development of data-parallel languages stemming from GPUs plays a crucial role when compiling them for a SIMD CPU: On the CPU, one has

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 56–70, 2013.

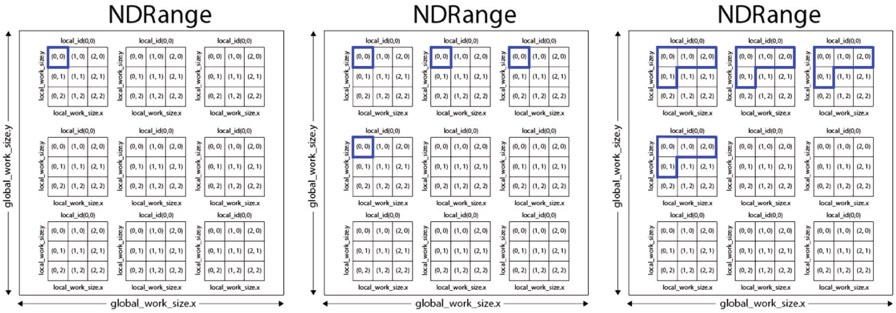


Fig. 1. The OpenCL execution model and examples how a driver could choose to execute work items in parallel (work items marked blue). Left: sequential execution. Middle: multi-threaded execution (4 cores). Right: multi-threaded execution with "Whole-Function Vectorization" (4 cores, SIMD width 4, details in Sect. 2).[1]

to emulate dynamic features that on GPUs are implemented in hardware. The interesting feature for this paper is that GPUs determine at runtime whether or not all work items access memory at *consecutive* addresses. In the positive case, a faster operation is issued. This behavior can be emulated by a CPU compiler that exploits SIMD instructions. To this end, it would introduce code that does the same runtime check, but the cost of this usually outweighs the performance gain. Thus, static analysis has been used to prove at compile time that a memory operation *always* accesses consecutive addresses [10]. This approach generally covers fewer cases than a dynamic runtime check but is still applicable often enough to be of interest. Until now, however, it could only handle very simple address computations such as linear translations by constants. In this paper, we generalize this to more interesting linear arithmetic transformations, in particular including integer division and modulo by constants. Our approach can—to a certain degree—also handle non-constant inputs. Our key idea is to convert the central question "Do consecutive work items access consecutive memory addresses or not?" to a finite set of satisfiability problems in *Presburger Arithmetic.*

Presburger Arithmetic originally refers to the first-order theory of the integers over a countably infinite language $\mathcal{L}$ comprising 0, 1, $+$, $-$, $<$, and infinitely many congruences $\equiv_k$ for $k \in \mathbb{N} \setminus \{0\}$. This, of course, allows to tacitly use also $\leq$, $\neq$, $>$, $\geq$. For this setting, Presburger proved completeness by giving a decision procedure [16]. His decision procedure was based on effective quantifier elimination in combination with the fact that variable-free atomic formulas can be effectively evaluated to either "true" or "false." As a consequence of using quantifier elimination, Presburger required $\equiv_k$ to be a formal part of the language.

[1] Graphics modified from the official Khronos OpenCL specification.

In the context of programming languages, Presburger's congruence relations have a counterpart in the binary modulo function, which is naturally paired with integer division. For fixed integer second argument $k \in \mathbb{N} \setminus \{0\}$, both can be encoded in $\mathcal{L}$, e.g., as follows:

$$\mathbb{Z} \models \mathrm{mod}_k(x) = y \longleftrightarrow 0 \leq y \leq k - 1 \land x \equiv_k y,$$
$$\mathbb{Z} \models \mathrm{div}_k(x) = y \longleftrightarrow k \odot y \leq x < k \odot (y + 1).$$

These defining formulas can be generalized to $k \in \mathbb{Z} \setminus \{0\}$ and can be used to systematically translate formulas containing mod_k and div_k to the original Presburger language $\mathcal{L}$. This requires, in general, the introduction of quantifiers with variables that represent sub-terms. Similarly, for decision procedures not based on quantifier elimination, the congruences can be eliminated:

$$\mathbb{Z} \models t \equiv_k 0 \longleftrightarrow \exists x (k \odot x = t).$$

Note that admitting arbitrary terms as second arguments in modulo operations or integer division would lead to an undecidable theory:

$$\mathbb{Z} \models s \neq 0 \longrightarrow t \bmod s = t - (t/s)s, \quad \mathbb{Z} \models s \mid t \longleftrightarrow t \bmod s = 0,$$

and $(\mathbb{Z}, 0, 1, +, -, |)$ is known to be undecidable [17].

There is a variety of decision procedures and complexity results available for Presburger Arithmetic [20, and the references given there]. Our input considered here is limited to the existential fragment, for which SMT solvers, in spite of their possible incompleteness, are an interesting choice. For our practical computations we chose Z3 [5], which has the advantage to directly accept mod_k and div_k in the input.

The original contributions of this paper are the following:

1. We formalize in Presburger Arithmetic the notion of consecutivity and all relevant conditions that have to be decided for static optimization of memory operations.
2. Our formalization allows to consider address computations that involve div_k and mod_k and limited occurrences of non-constant inputs.
3. We introduce *modulo elimination* as a general preprocessing technique for Presburger terms. This makes it feasible to decide our formalizations using an off-the-shelf SMT solver.
4. The feasibility of our approach is documented by comprehensive, systematic computations.
5. Our computations establish new benchmarks based on current research problems in compiler construction. In this capacity, they are of general interest for the SMT community.
6. We show that a prototypical OpenCL CPU driver based on our approach generates more efficient code than any other state-of-the-art driver, including Intel and AMD.

```
__kernel void
simple(float* in,
       float* out) {
  int tid = get_global_id();
  out[tid] = in[tid];
}
```

```
__kernel void
fastWalshTransform(float* tArray,
                   int    step) {
  int tid   = get_global_id();
  int group = tid % step;
  int pair  = 2*step*(tid/step)
            + group;
  int match = pair + step;
  float T1       = tArray[pair];
  float T2       = tArray[match];
  tArray[pair]   = T1 + T2;
  tArray[match]  = T1 - T2;
}
```

Fig. 2. OpenCL kernels with simple (left) and complex (right) memory address computations: `tid`, `pair`, and `match`. The function `get_global_id()` returns the work item ID, which allows each work item to access different array positions.

The structure of the paper is as follows: In Sect. 2, we summarize the relevant notions from data-parallel languages and compilers and make precise the problem addressed in this paper. In Sect. 3, we formalize this problem as a set of Presburger satisfiability problems and perform first computational experiments. Sect. 4 details how modulo elimination significantly improves performance of the SMT solver. In Sect. 5, we discuss possibilities for code generation. Sect. 6 experimentally evaluates the achieved performance gain. Sect. 7 discusses related work. In Sect. 8, we summarize our results and discuss possible future work.

2 Memory Access Optimization for Data-Parallel Languages

We want the compiler of a language like CUDA or OpenCL to prove for as many memory access operations as possible that the addresses that are accessed by consecutive work items are contiguous in memory. If this property can be proven, CPU compilers for these languages can generate faster code. Recall from the introduction that our target architecture are CPUs with SIMD instruction sets, to which we are going to simply refer to as CPUs. We generally consider load and store operations, for which array accesses are the most prominent example.

Consider the two OpenCL kernels in Fig. 2. The kernel on the right-hand side, `fastWalshTransform`, is taken from the AMD APP SDK v2.8.[2] In this code, the array accesses depend on the value `tid` obtained from calls to `get_global_id()`. These calls return different values for different work items and consecutive values for consecutive work items. It is easy to see that the `simple` kernel always accesses contiguous memory locations due to the direct

[2] `developer.amd.com/sdks/AMDAPPSDK`

```
__kernel void
fastWalshTransform(float* tArray,
                   int    step)
{
  int tid    = get_global_id();
  if (tid % 2 != 0) return;
  int2 tidV  = (int2)(tid,tid+1);
  int2 stepV = step;
  int2 group = tidV % stepV;
  int2 pair  = 2*step*(tidV/stepV)
                + group;
  int2 match = pair + stepV;
  float2 T1  = (float2)(tArray[pair.x],
                        tArray[pair.y]);
  float2 T2  = (float2)(tArray[match.x],
                        tArray[match.y]);
  float2 X   = T1 + T2;
  float2 Y   = T1 - T2;
  tArray[pair.x]  = X.x;
  tArray[pair.y]  = X.y;
  tArray[match.x] = Y.x;
  tArray[match.y] = Y.y;
}
```

```
__kernel void
fastWalshTransform(float* tArray,
                   int    step)
{
  if (step <= 0 || step % 2 != 0) {
    // Omitted code:
    // Execute kernel as on the left.
    return;
  }
  int tid   = get_global_id();
  if (tid % 2 != 0) return;
  int group = tid % step;
  int pair  = 2*step*(tid/step)
            + group;
  int match = pair + step;
  float2 T1 = *((float2*)(tArray+pair));
  float2 T2 = *((float2*)(tArray+match));
  *((float2*)(tArray+pair))  = T1 + T2;
  *((float2*)(tArray+match)) = T1 - T2;
}
```

Fig. 3. Result of WFV manually applied at source level to the `fastWalshTransform` kernel of Fig. 2 ($w = 2$). Left: Conservative WFV requires sequential execution. Right: WFV with our approach proves consecutivity of the memory addresses for certain values of `step`, which allows to generate a variant with more efficient code.

use of `tid`. In contrast, the access pattern of `fastWalshTransform` is not obvious. Experimentally, one would observe that, depending on `step`, there is a considerable number of accesses that actually are consecutive. At this point, it is important to understand a fundamental difference between GPUs and CPUs: In GPUs, there is dedicated hardware to dynamically *coalesce* to a single access all memory accesses of work items running in parallel whenever possible. This yields significant speedup by preventing sequential execution of the accesses. On CPUs, there is no such hardware support. Therefore, without compiler optimization, the memory operations considered here would be executed sequentially.

Modern compilers apply a technique called *Whole-Function Vectorization* (WFV) [10]. WFV transforms a kernel to execute w work items at once, where w is usually the *SIMD width*. The SIMD width of a CPU is the number of single-precision values that fit into a vector register. Typical values for w are 4 for the SSE, AltiVec, and NEON instruction sets, 8 for AVX, or 16 for LRBni. An interesting recent development is AMDs introduction of the Sea Islands series of GPUs which have a vector instruction set with a SIMD width of 64. In the context of WFV, a vectorized kernel executes w work items at once for every single hardware thread. The values of each work item are kept in one cell of a vector register instead of a scalar register. Thus, WFV can increase performance of applications by a factor as large as w.

Unfortunately, WFV has drawbacks which can significantly reduce this theoretical speedup or even result in slowdowns [11]. In this paper, we focus on one specific drawback, which arises in presence of memory access operations: In order to have the kernel compute w work items at once, accesses to `tid` are transformed to return a vector of w consecutive values. Accordingly, each dependent operation has to be transformed into its vector counterpart, e.g. a scalar addition becomes a vector addition. Unfortunately, the vector counterparts for memory operations available in most of today's SIMD instruction sets only support accessing consecutive addresses. Therefore, if the addresses are non-consecutive, w sequential operations have to be used, reducing the overall gain of WFV compared to the theoretical factor w. This is exemplified by the left-hand side kernel in Fig. 3, where `tidV` is the above-mentioned, vector-valued `tid`. To make use of a vector load or store, the compiler has to automatically prove that the address computation will never produce non-consecutive values.[3] However, the expression tree that corresponds to this address computation may consist of arbitrary code. This will, in general, lead to undecidable problems. Current approaches are limited to expressions with linear translations by constants [10]. Our new approach extends the class of expressions that can be analyzed to Presburger Arithmetic and functions definable therein. This covers in particular integer division and modulo operations by constants as well as certain occurrences of input variables.

To this end, our compiler translates the expression tree that yields the memory address to a term that depends on `tid` and a possible input parameter. For example, the address of the second load operation of the FastWalshTransform kernel in Fig. 2 is given by `tArray[match]`, where the term obtained for `match` is

```
2*step * (tid / step) + (tid % step) + step.        (1)
```

Notice that `step` is an input value that is constant for all work items during one execution of the kernel.

3 Translation to Presburger Arithmetic

We are now going to switch to a more mathematical notation: The variable t is going to denote the `tid` and a is going to denote the input. For integer division and modulo, we introduce unary functions div_k and mod_k for $k \in \mathbb{Z} \setminus \{0\}$, which emphasizes the fact that the divisors and moduli are limited to numbers. For our example term (1), we obtain

$$e(t, a) = 2a \odot \operatorname{div}_a(t) + \operatorname{mod}_a(t) + a. \tag{2}$$

At this point, let us give the precise definitions of mod_k and div_k:

$$x = k \odot \operatorname{div}_k(x) + \operatorname{mod}_k(x), \quad \text{where} \quad |\operatorname{mod}_k(x)| < |k|. \tag{3}$$

[3] In addition, a store must be proven to be always executed by all work items to not produce false side effects. To keep things simple, we consider this out of the scope of this paper.

It is well-known that this definition does not uniquely specify $\mathrm{div}_k(x)$ and $\mathrm{mod}_k(x)$. SMT-LIB Version 2 resolves this issue by making the convention that $\mathrm{mod}_k(x) \geq 0$.[4] As long as both k and x are non-negative, common programming languages agree with this convention. However, when negative numbers are involved, OpenCL follows the C99 standard, which in contrast to SMT-LIB requires that $\mathrm{sign}(\mathrm{mod}_k(x)) = \mathrm{sign}(x)$. In our setting, we observe that the arguments of mod_k generally are positive expressions involving the `tid` such that both conventions happen to coincide.

Let us analyze a single memory access with respect to the following *consecutivity question*: "Do w consecutive work items access consecutive memory addresses when doing this memory access or not?" Using the corresponding term $e(t, a)$, the following equation holds if and only if the work items t and $t+1$ access consecutive memory locations for input a:

$$e(t, a) + 1 = e(t + 1, a).$$

The following conjunction generalizes this equation to w consecutive work items t, $\dots$, $t + w - 1$:

$$\bigwedge_{i=0}^{w-2} e(t + i, a) + 1 = e(t + i + 1, a).$$

Recall from the previous section that these groups of w work items naturally start at 0 so that only conjunctions are relevant where t is divisible by w. The following Presburger formula formally adds this constraint:

$$\varphi(w, a) = \forall t\Big(t \geq 0 \land t \equiv_w 0 \longrightarrow \bigwedge_{i=0}^{w-2} e(t + i, a) + 1 = e(t + i + 1, a)\Big).$$

For given $w \in \mathbb{N}$ and $\alpha, \beta \in \mathbb{Z}$ with $\alpha \leq \beta - 1$, the answer to our consecutivity question for w and $a \in \{\alpha, \dots, \beta - 1\}$ is given by the set

$$A_{w,\alpha,\beta} = \{\, a \in \mathbb{Z} \mid \mathbb{Z} \models \varphi(w, a) \land \alpha \leq a < \beta \,\}.$$

We essentially compute $A_{w,\alpha,\beta}$ by at most $(w-1)(\beta - \alpha - 1)$ many applications of an SMT solver to the $w-1$ disjuncts of $\neg\varphi(w, a)$ for $a \in \{\alpha, \dots, \beta - 1\}$, where

$$\neg\varphi(w, a) = \bigvee_{i=0}^{w-2} \exists t\big(t \geq 0 \land t \equiv_w 0 \land e(t + i, a) + 1 \neq e(t + i + 1, a)\big).$$

Notice that, when obtaining "sat" for some $i \in \{0, \dots, w - 2\}$, the remaining problems of the disjunction need not be computed.

Our answer $A_{w,\alpha,\beta}$ consists of those a for which the SMT solver yields "unsat." Note that besides "sat" or "unsat," the solver can also yield "unknown," which we treat like "sat." This underapproximation does not affect the correctness of our approach. We only miss optimization opportunities when generating code

[4] `smtlib.cs.uiowa.edu/theories/Ints.smt2`

Table 1. Running times of Z3 applied to $\neg\varphi(w, a)$ for $e(t, a)$ as in (2). In all three cases, $\alpha = 1$ and $\beta = 2^{16}$ so that $a \in \{1, \ldots, 2^{16} - 1\}$ with a time limit of one minute per call.

FastWalshTransform Problem Set using Z3

w	Sat	Unsat	Unknown	Timeouts	CPU Time
4	16,243	48,931	0	361	14 h
8	7,694	54,510	0	3,331	97 h
16	2,773	52,468	0	10,294	256 h

later on. The same holds for possible timeouts when imposing reasonable time limits on the single solver calls. Later in Sect. 5, we are going to discuss how compact representations for $A_{w,\alpha,\beta}$ can be obtained.

Table 1 shows running times and results for the application of Z3 version 4.3.1 [5] to the consecutivity question for our FastWalshTransform kernel.[5] Alternatives to Z3 include CVC4 [1] and MathSAT5 [3]. These SMT solvers, however, do not directly support div_k and mod_k, which makes them less interesting for our application here.

Obviously, our obtained running times are too high to be of any practical interest. We suspect that the div_a and mod_a operators occurring in (2), which have to be resolved into the classical Presburger language, significantly contribute to that complexity.

4 Modulo Elimination as a Preprocessing Step

We are now going to considerably improve the running times observed in the previous section. The basic idea is to eliminate, in a preprocessing step, all occurrences of mod_k in favor of div_k. To this end, we use the definition (3) of div_k and mod_k as a rewrite rule:

$$\mathrm{mod}_k(x) \to x - k \odot \mathrm{div}_k(x).$$

The validity of this *modulo elimination* rule is not affected by the discussion after the statement of definition (3). After finitely many applications of the rule to $\neg\varphi(w, a)$, we arrive at equivalent input to the SMT solver that does not contain any modulo operations.

Applying modulo elimination to $e(t, a)$ from (2) results in $a + t + a \odot \mathrm{div}_a(t)$. Using this reduced term instead of the original one inside $\neg\varphi(w, a)$ makes a significant difference in performance for the benchmarks from Table 1. The improved results are collected in Table 2. Notice that the CPU times related by the respective speedup factors refer to different subsets of finished computations due to timeouts. Nevertheless, for practical purposes, these speedups are exactly the

[5] All our SMT computations have been performed on a 2.4 GHz Intel Xeon E5-4640 running Debian Linux 64 bit.

Table 2. Running times of Z3 applied to $\neg\varphi(w,a)$ for $e(t,a)$ obtained by applying modulo elimination to the term in (2). In all three cases, $\alpha = 1$ and $\beta = 2^{16}$ so that $a \in \{1, \dots, 2^{16} - 1\}$ with a time limit of one minute per call.

FastWalshTransform Problem Set with Modulo Elimination using Z3

w	Sat	Unsat	Unknown	Timeouts	CPU Time	Speedup
4	16,383	49,152	0	0	4 min	210×
8	8,191	57,344	0	0	5 min	1164×
16	4,095	61,128	0	312	334 min	46×

interesting information. We have verified that our modulo elimination causes at least 50 percent of the reported speedups.

These running times are not suitable for just-in-time compilation. For offline release compilation, however, they are fine. Note that the computations can be perfectly parallelized such that the use of, say, 64 virtual cores will result in a corresponding speedup factor. We estimate that on such a system, the FastWalshTransform problem set for $w = 4$ could be computed in less than seventy hours even for $\alpha = -2^{31}$ and $\beta = 2^{31}$ covering the full range of signed 32 bit integers. In general, there are situations where an optimization for a subset of the range will result in a good tradeoff between compilation time and performance gain.

Let us understand why modulo elimination is so successful on our types of examples: The relevant equation $e(t+1,a) - e(t,a) - 1 = 0$ for FastWalshTransform with $e(t,a)$ taken from (2) is given by

$$2a \odot \mathrm{div}_a(t+1) - 2a \odot \mathrm{div}_a(t) + \mathrm{mod}_a(t+1) - \mathrm{mod}_a(t) - 1 = 0.$$

Modulo elimination yields:

$$2a \odot \mathrm{div}_a(t+1) - 2a \odot \mathrm{div}_a(t) + t + 1 - a \odot \mathrm{div}_a(t+1) - t + a \odot \mathrm{div}_a(t) - 1 = 0,$$

which simplifies to $a \odot \mathrm{div}_a(t+1) - a \odot \mathrm{div}_a(t) = 0$. Such patterns do not occur accidentally: Programmers using data-parallel languages try hard to access memory only consecutively to get the best performance. This leads to specific patterns within address computation expressions. Using particularly div_k and mod_k, consecutivity can hardly be achieved without patterns similar to the one discussed above.

To conclude this section, we are now going to discuss an additional example, for which our approach turns out to be suitable even for just-in-time compilation. This is a kernel, called `bitonicSort`, also taken from the AMD APP SDK. The expression we are interested in here is

$$e(t,a) = 2^{a+1} \odot \mathrm{div}_{2^a}(t) + \mathrm{mod}_{2^a}(t) + 2^a. \tag{4}$$

The input parameter a occurs exclusively as an exponent. This restricts the reasonable range of values to consider to $\{0, \dots, 62\}$ on a 64 bit architecture.

Table 3. Running times of Z3 applied to $\neg\varphi(w, a)$ for $e(t, a)$ as in (4). In all three cases, $\alpha = 0$ and $\beta = 63$ so that $a \in \{0, \dots, 62\}$ with a time limit of one minute per call.

BitonicSort Problem Set using Z3

w	Sat	Unsat	Unknown	Timeouts	CPU Time
4	54	2	0	7	8 min
8	52	3	0	8	16 min
16	40	4	0	19	41 min

Table 4. Running times of Z3 applied to $\neg\varphi(w, a)$ for $e(t, a)$ obtained by applying modulo elimination to the term in (4). In all three cases, $\alpha = 0$ and $\beta = 63$ so that $a \in \{0, \dots, 62\}$ with a time limit of one minute per call.

BitonicSort Problem Set with Modulo Elimination using Z3

w	Sat	Unsat	Unknown	Timeouts	CPU Time	Speedup
4	61	2	0	0	0.7 s	686×
8	60	3	0	0	1.5 s	640×
16	59	4	0	0	3.7 s	665×

Table 4 shows the relevant running times. Noticing the similarities between (4) and (2), it is clear that modulo elimination leads to similar simplifications in the corresponding equation. The timings in Table 3 confirm this: The speedups are similar to FastWalshTransform.

5 From SMT Solving Results to Code

Recall from Sect. 3 that the answer obtained there to our consecutivity question "Do w consecutive work items access consecutive memory addresses when doing this memory access or not?" is the set $A_{w,\alpha,\beta}$ of inputs $a \in \{\alpha, \dots, \beta - 1\}$ for which the answer is affirmative. The respective sets $A_{w,\alpha,\beta}$ for all our problem sets are collected in Table 5.

Our goal is now to produce during code generation a case distinction such that for input contained in $A_{w,\alpha,\beta}$, more efficient code including vector memory operations will be executed. The right-hand side of Fig. 3 shows the automatically generated code for the `fastWalshTransform` kernel for $w = 2$ without imposing bounds α and β. For readability reasons, we use OpenCL notation instead of the LLVM intermediate representation [13], which we actually use at that stage of compilation. The corresponding condition

```
step <= 0 || step % 2 != 0                    (5)
```

in the first if-statement describes the complement of the set $A_{w,\alpha,\beta}$ obtained from our SMT solving step.

Table 5. Output from the SMT solving step for all our problem sets. We have $X \subseteq \{ a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_{16} 0 \}$ with $|X| = 312$, i.e., timeouts occur only for input a with $a \equiv_{16} 0$.

SMT Solving Step Output

Problem Set	w	α	β	$A_{w,\alpha,\beta}$
FastWalshTransform	4	1	2^{16}	$\{ a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_4 0 \}$
FastWalshTransform	8	1	2^{16}	$\{ a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_8 0 \}$
FastWalshTransform	16	1	2^{16}	$\{ a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_{16} 0 \} \setminus X$
BitonicSort	4	0	63	$\{0, \dots, 62\} \setminus \{0, 1\}$
BitonicSort	8	0	63	$\{0, \dots, 62\} \setminus \{0, 1, 2\}$
BitonicSort	16	0	63	$\{0, \dots, 62\} \setminus \{0, 1, 2, 3\}$

Due to our independent runs of the SMT solver for all possible choices of a, the sets $A_{w,\alpha,\beta}$ are obtained explicitly as lists of elements. From these, we have to generate implicit descriptions like (5). One approach for this is to represent the characteristic functions of the sets $A_{w,\alpha,\beta}$ as bit strings and to use incremental finite automata minimization to obtain minimal regular expressions. These are finally transformed into quantifier-free Presburger conditions. Alternatively, one could apply automatic synthesis techniques as suggested by Gulwani et al. [7]. At present, this step is not automated yet.

6 Evaluation: OpenCL Performance

We evaluate the effect of our improved code generation for memory accesses for the applications that contain the kernels we discussed throughout the paper: FastWalshTransform and BitonicSort. To this end, we have hooked into the WFV OpenCL driver [11], employing our SMT-based approach to generate machine code.

In each respective kernel, there are actually several memory operations, which happen to lead to the same satisfiability problems $\neg\varphi(w, a)$. It is worth noting that for the majority of kernels that we found in the AMD APP SDK, the memory address computations are so simple that the relevant equations are decided already via term simplification, i.e. without any non-trivial SMT solving. Nevertheless, our technique is to our knowledge the first one that enables the compiler to generate better code in less simple cases such as FastWalshTransform and BitonicSort. Hence, if maximum performance of a kernel with complex memory operations is desired, our approach is the only currently available option.

In Table 6, we report kernel execution times of our SMT-enhanced driver in different configurations.[6] Each measurement shows the median of 1000 individual

[6] These experiments were conducted on a Core 2 Quad at 2.8 GHz with 8 GB of RAM running Ubuntu Linux 64 bit. The vector instruction set is Intel's SSE 4.2, yielding a SIMD width of four 32 bit values.

Table 6. Median of kernel execution times for 1000 executions of various OpenCL CPU drivers: Non-vectorized (Scalar), vectorized (WFV), and vectorized with our SMT-based optimization (WFV+SMT). As a reference, we also include the performance of the latest Intel and AMD implementations.[7] The Speedup column shows the effect of our SMT-based memory access optimization, comparing WFV+SMT to WFV.

OpenCL Kernel Performance (milliseconds)

Application	Array Size	AMD	Intel	Scalar	WFV	WFV+SMT	Speedup
FastWalshTransform	16,777,216	413	313	303	309	299	1.03×
BitonicSort	1,048,576	894	680	236	121	58	2.09×

runs per configuration per benchmark without warm-up. Although the machine was not rebooted after every run, the numbers reported here are as realistic as possible for one cold-started, arbitrary run of the application.

The results clearly demonstrate the applicability of our approach. For FastWalshTransform, this is the first time that we were able to beat the scalar implementation with the WFV-based one [11]. It turns out, however, that the optimized code can be executed in only one out of w cases, which limits the performance gain. The situation is different for BitonicSort. Here, the optimized code is executed in the majority of cases. The factor of 2.09 for BitonicSort is huge, and also the 3 percent speedup for FastWalshTransform is relevant.

It is remarkable that in spite of including a WFV implementation, the Intel driver *refuses* to vectorize either of the two kernels. This means that Intel's heuristics deem the code to not benefit from vectorization. The reasons are probably that they consider the memory operations to dominate the runtime and that the heuristics have to assume that these operations are not consecutive. The AMD CPU driver does not use WFV.

7 Related Work

The basic analysis of memory address computations for linear translations by constants, which we extend, was introduced as part of the *vectorization analysis* of WFV [10,11]. Coutinho et al. [4] proposed a similar *divergence analysis*, which identifies values that remain the same for all work items but does not analyze consecutivity.

An increasing number of OpenCL drivers is being developed by different software vendors for all kinds of platforms from GPUs to mobile devices. For comparison purposes, the x86 CPU compiler from Intel is most interesting, although most details about the underlying implementation are not disclosed. According to our experimental results, its analyses are not as advanced as ours. The Portland Group implemented an x86 CPU driver for CUDA, which also makes use of

[7] `software.intel.com/en-us/vcsource/tools/opencl`, the Intel SDK for OpenCL Applications XE 2013 Beta.

both multi-threading and WFV.[8] Again, no details are publicly available. The AMD driver, the POCL project [9], TwinPeaks [8], MCUDA [18], and Ocelot [6] are other notable CPU implementations of the OpenCL and CUDA APIs, but none of them employ WFV.

For GPUs, various approaches exist to analyze memory access patterns for coalescing. However, none of the static approaches can handle integer division, modulo by constants, or non-constant inputs. CuMAPz [12] and the official CUDA Visual Profiler perform dynamic analyses of memory access patterns and report non-coalesced operations to the programmer. Yang et al. [21] implemented a static compiler optimization to improve non-coalesced accesses using shared memory. Li et al. [14] proposed an SMT-based approach for verification of GPU kernels. This was extended by Lv et al. [15] to also profile coalescing. Tripakis et al. [19] use an SMT solver to prove non-interference of SPMD programs. GPUVerify [2] is a tool that uses Z3 to prove OpenCL and CUDA kernels to be free from race-conditions and barrier divergence. None of these SMT-based techniques is concerned with automatic code optimization but only with verification.

8 Conclusions and Future Work

We have improved the state-of-the art in CPU code generation for memory access operations in data-parallel languages. The key idea is to prove at compile time that certain memory operations access consecutive addresses. This task is automatically translated to sets of formal problems that can be processed by an off-the-shelf SMT solver. We have introduced modulo elimination, a preprocessing technique on those formal problems which makes the approach practically feasible. The SMT output admits to construct case distinctions that are crucial for generating efficient code. Our performance measurements have demonstrated that our generated code is more efficient than all previous approaches including proprietary implementations by Intel and AMD.

To turn our prototypical environment into one integrated software system, a few gaps have to be closed: The compiler has to be linked with the SMT solver to minimize communication overhead. The code generation via finite automata or automatic synthesis as suggested in Sect. 5 has not yet been implemented.

On the other hand, the results achieved here open up interesting perspectives for future work in the areas combined in this paper: Modulo elimination should be included in SMT solvers with appropriate heuristics. More generally, sophisticated translation of integer division and modulo operations in special cases might boost performance also in other application areas. It appears promising to generate from the final SMT input $\neg\varphi(w, a)$ equivalent integer linear programs and to compare the performance of corresponding software. On the compiler side, a next step would be to automatically deduce tight ranges for the input values, e.g., from their data types or even from their usage. It is quite clear that our approach can be adapted to GPUs with SIMD instruction sets that have recently entered the market, e.g., the latest AMD Sea Islands series.

[8] The Portland Group, Inc. PGI CUDA-x86.

Acknowledgements. We would like to thank Sebastian Hack for helpful discussions. This research was supported in part by the German Transregional Collaborative Research Center SFB/TR 14 AVACS and in part by the German Federal Ministry of Education and Research (BMBF).

References

1. 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)
2. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: Gpuverify: a verifier for gpu kernels. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, pp. 113–132. ACM, New York (2012)
3. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
4. Coutinho, B., Sampaio, D., Pereira, F.M.Q., Meira, W.: Divergence analysis and optimizations. In: 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 320–329 (2011)
5. 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)
6. Diamos, G.F., Kerr, A.R., Yalamanchili, S., Clark, N.: Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, pp. 353–364. ACM, New York (2010)
7. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 62–73. ACM, New York (2011)
8. Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B.R., Zheng, B.: Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In: PACT, pp. 205–216. ACM, New York (2010)
9. Jaskelainen, P.O., de La Lama, C.S., Huerta, P., Takala, J.: OpenCL-based Design Methodology for Application-Specific Processors. In: SAMOS 2010, pp. 223–230 (July 2010)
10. Karrenberg, R., Hack, S.: Whole function vectorization. In: CGO, pp. 141–150 (2011)
11. Karrenberg, R., Hack, S.: Improving Performance of OpenCL on CPUs. In: O'Boyle, M. (ed.) CC 2012. LNCS, vol. 7210, pp. 1–20. Springer, Heidelberg (2012)
12. Kim, Y., Shrivastava, A.: Cumapz: a tool to analyze memory access patterns in CUDA. In: Proceedings of the 48th Design Automation Conference, DAC 2011, pp. 128–133. ACM, New York (2011)
13. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO (March 2004)
14. Li, G., Gopalakrishnan, G.: Scalable smt-based verification of GPU kernel functions. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2010, pp. 187–196. ACM, New York (2010)

15. Lv, J., Li, G., Humphrey, A., Gopalakrishnan, G.: Performance degradation analysis of gpu kernels. In: Proceedings of the Workshop on Exploiting Concurrency Efficiently and Correctly 2011, EC2 2011 (2011)
16. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes Rendus du Premier Congres de Mathematiciens des Pays Slaves, Warsaw, Poland, pp. 92–101 (1929)
17. Robinson, J.: Definability and decision problems in arithmetic. J. Symb. Log. 14(2), 98–114 (1949)
18. Stratton, J.A., Stone, S.S., Hwu, W.-M.W.: MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 16–30. Springer, Heidelberg (2008)
19. Tripakis, S., Stergiou, C., Lublinerman, R.: Checking equivalence of spmd programs using non-interference. Technical Report UCB/EECS-2010-11, EECS Department, University of California, Berkeley (January 2010)
20. Weispfenning, V.: The complexity of almost linear Diophantine problems. Journal of Symbolic Computation 10(5), 395–403 (1990)
21. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, pp. 86–97. ACM, New York (2010)

Roughening the $\mathcal{EL}$ Envelope

Rafael Peñaloza[1,⋆] and Tingting Zou[2]

[1] Theoretical Computer Science, TU Dresden
Center for Advancing Electronics Dresden, Germany
penaloza@tcs.inf.tu-dresden.de
[2] College of Computer Science and Technology, Jilin University, China
zoutingt@163.com

Abstract. The $\mathcal{EL}$ family of description logics (DLs) has been successfully applied for representing the knowledge of several domains, specially from the bio-medical fields. One of its principal characteristics is that its reasoning tasks have polynomial complexity, which makes them suitable for large-scale knowledge bases. In their classical form, description logics cannot handle imprecise concepts in a satisfactory manner. Rough sets have been studied as a method for describing imprecise notions, by providing a lower and an upper approximation, which are defined through classes of indiscernible elements.

In this paper we study the combination of the $\mathcal{EL}$ family of DLs with the notion of rough sets, thus obtaining a family of rough DLs. We show that the rough extension of these DLs maintains the polynomial-time complexity enjoyed by its classical counterpart. We also present a completion-based algorithm that is a strict generalization of the known method for the DL $\mathcal{EL}^{++}$.

1 Introduction

Description Logics (DLs) [3] are a family of knowledge representation formalisms designed for expressing terminological knowledge in an unambiguous and well-understood manner. They have been successfully applied to modelling and reasoning with real-world knowledge domains, but arguably its largest success so far is the designation of the DL-based language OWL as the standard ontology language for the semantic web, by the W3C.[1]

The DL $\mathcal{EL}$ is a lightweight logic that allows only for conjunction and existential restrictions as constructors. As it cannot express negations, $\mathcal{EL}$ is not propositionally closed. Despite its low expressivity, this logic and small extensions of it have been successfully used for representing knowledge from several domains, most prominently from the medical and biological fields. In fact, minor extensions of $\mathcal{EL}$ are the basic logics underlying large-scale ontologies like Snomed CT[2] or the Gene Ontology.[3] A prominent feature of these logics is

[⋆] Partly supported by DFG within the Cluster of Excellence 'cfAED'.

[1] http://www.w3.org/TR/owl2-overview/

[2] http://www.ihtsdo.org/snomed-ct/

[3] http://www.geneontology.org

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 71–86, 2013.

their polynomial-time complexity of reasoning, which enables effective reasoning procedures. In fact, modern reasoners are capable of classifying SNOMED CT, which has approximately 300,000 axioms, in less than seven seconds [16].

In their classical form the members of the $\mathcal{EL}$ family, as all other classical DLs, lack the capacity of modelling and reasoning with imprecise knowledge. This is in no way a small drawback, as imprecision is almost unavoidable in several knowledge domains, like those from the bio-medical fields. For example, even the notion of *species*, one of the mayor taxonomic ranks from biological classification is far from precise, or even being well-understood. Consider for instance the case of the *Ensatina* salamanders from North America. When seen independently, the Monterey Ensatina and the Large Blotched Ensatina form two different species, with their own characteristic traits; they can be easily distinguished as the former is completely brown in color, while the latter is black with large yellow blotches. Moreover, these two groups of individuals are uncapable to interbreed, which is the minimal requirement for distinguishing elements of a species. However, there also exists a group of *intermediate* individuals, that mix the traits of both species, forming a gradual bridge between them; e.g., dark brown with lighter-brown blotches. These intermediate individuals form also a chain of interbreeding relations that goes from the Monterey to the Large Blotched Ensatinas. It is thus unclear at which point these intermediate individuals stop being members of one species and start belonging to the other. Indeed, providing a satisfactory notion of when two individuals belong to the same species is a prominent problem in biology [11].

The best-known approach for handling imprecision formally is through fuzzy logic [13]. Fuzzy extensions of DLs have been thoroughly studied during the last decade as a formalism for representing vague terminological knowledge [19,23]. However, it was recently shown that reasoning in expressive fuzzy DLs is either undecidable [6,9], or must ignore the truth degrees [5]. Even for the inexpressive DL $\mathcal{EL}$, the extension to general fuzzy-set based semantics usually yields intractable reasoning problems [8]. It can be argued that these negative complexity results arise from the high level of granularity provided by fuzzy semantics, where every number from the interval $[0, 1]$ can be used as a truth degree. In other words, it is possible to make arbitrarily small distinctions between elements of the domain. One can partially alleviate this problem by restricting to finitely many truth degrees [4,7]. In this case, the resources needed for reasoning are directly correlated with the size of the truth value space. This idea, however, adds the burden of deciding *a priori* the amount of degrees that will be needed and their relevant operations. It is thus desirable to obtain an intermediate formalism that allows for imprecise limitations of concepts, while avoiding the level of detail of fuzzy logics.

Rough sets were introduced in [20] as an alternative to fuzzy set theory [26] for dealing with imprecise notions. The main idea behind this formalism is to describe imprecise sets by allowing a class of *boundary* elements that can neither be stated to belong, nor to be outside, the set. More precisely, a set X without a clear distinction on its limits, is approximated using a set $\underline{X}$ of elements that are guaranteed to belong to X, and a set $\overline{X}$ of elements that might be members of X; this latter set is called the upper approximation of X. These sets are formally defined with the help of an *indiscernibility relation* that clusters together individuals sharing the same properties. The difference $\overline{X} \setminus \underline{X}$ are the boundary elements, which cannot be ensured to belong to X, nor to its complement.

For example, the problem with the different species of Ensatina salamanders can be solved by stating that the intermediate individuals belong to the upper bounds of the sets of both species. This representation allows us to state properties of the intermediate individuals (e.g. that they have mixed traits from the border species) without providing a clear-cut division of these individuals into the two species.

In this paper we study rough $\mathcal{EL}^{++}$, a logic that combines the DL $\mathcal{EL}^{++}$ (without concrete domains) and rough set semantics. Although the combination of rough set theory with DLs is far from new (see e.g. [18] for some early work), interest in it has grown in the last few years [10,14,17,22]. Most of the work in this direction so far focuses on rough extensions of expressive DLs. The approach is to extend a description logic with two new constructors that describe the upper and lower approximations of concepts. The semantics of these constructors are based on equivalence relations that provide the indiscernibility relation from rough set theory. In [22] it was shown that these constructors can be modelled in classical DLs with the help of existential and value restrictions over a new transitive, symmetric and reflexive role ρ. Briefly, the role ρ describes the indiscernibility relation, and the value and existential restrictions can be used to describe the lower and upper approximations, respectively. This construction is useful for showing that the rough constructors do not increase the complexity of standard reasoning for expressive DLs.

The reduction from [22], when applied to rough $\mathcal{EL}^{++}$, requires to extend the set of constructors to include value restrictions and inverse roles, among others. The extensions of $\mathcal{EL}^{++}$ with any of these constructors are known to be ExpTime-complete [1,2]. Thus, this approach yields an exponential-time upper bound for reasoning in rough $\mathcal{EL}^{++}$, in contrast to the polynomial-time complexity for classical $\mathcal{EL}^{++}$. In this paper we show that subsumption in rough $\mathcal{EL}^{++}$ is in fact PTime-complete, matching the known complexity for its classical logic.

The paper is divided as follows. We first provide a very brief introduction to the theory of rough sets, which will be useful for defining the syntax and semantics of rough $\mathcal{EL}^{++}$ in Section 3, where we also prove some basic properties of this logic. In Section 4, we describe a completion-based algorithm for deciding subsumption of rough $\mathcal{EL}^{++}$ concepts. As an added benefit, we obtain that *classifying* the full ontology needs only polynomial time. This paper extends the results from [21].

2 Rough Sets

Rough sets were introduced in [20] as an alternative to fuzzy set theory [26] for dealing with imprecise notions. The main motivation in this formalism is to be able to *approximate* terms that defy a precise characterisation, with the help of an equivalence relation $\sim$, called the *indiscernibility relation.* Formally, the equivalence relation $\sim$ divides the universe into its equivalence classes, which form clusters, or *granules* of indiscernible elements. Intuitively, elements belonging to the same equivalence class cannot be distinguished through their perceivable characteristics, and hence cannot be divided by a given set. Rough sets are also sometimes called granular sets in the literature and are one of the basis for granular computing [25].

Given a set X, and an equivalence relation $\sim$, we can define its *best lower approximation*, denoted by $\underline{X}$, as the greatest union of equivalence classes contained in X; i.e., $\underline{X} := \bigcup_{[x]_\sim \subseteq X} [x]_\sim$. Likewise, its *best upper approximation* is the union of the equivalence classes of all elements of X; $\overline{X} := \bigcup_{x \in X} [x]_\sim$. Equivalently, we have

$$\underline{X} = \{x \mid [x]_\sim \subseteq X\}, \qquad \overline{X} = \{x \mid [x]_\sim \cap X \neq \emptyset\}.$$

The elements in $\underline{X}$ are those that can be clearly distinguished from any element not belonging to X, and hence are said to *surely* belong to X. The members of $\overline{X}$, on the other hand, are those indistinguishable from some element of X, and said to *possibly* belong to X. The elements in the *boundary* $\overline{X} \setminus \underline{X}$ of X are those for which the notion of *belonging to* X cannot be made precise, as they are indistinguishable from both, members X and members of the complement of X.

From an informal point of view, it is possible to see rough sets as a three-valued membership function, where members of $\underline{X}$ *strongly belong* to X, the boundary elements *weakly belong* to X, and those in the complement of $\overline{X}$ do not belong to X. However, this description is overly simplistic, as the three-valued semantics are incapable of fully characterising the properties of the indiscernibility relation. In particular, the desired properties relating a three-valued conjunction with its three-valued implication cannot be enforced through the conjunction and implication of rough sets.

In the next section, we describe the combination of the description logic $\mathcal{EL}$ with the lower and upper-approximation constructors, whose semantics is based on rough sets. Afterwards, we describe a completion algorithm for deciding (classical) subsumption between rough $\mathcal{EL}$ concepts.

3 Rough $\mathcal{EL}^{++}$

The logic rough $\mathcal{EL}^{++}$ extends classical $\mathcal{EL}^{++}$ by allowing the *lower approximation* and *upper approximation* constructors $\underline{\ }$ and $\overline{\ }$ for expressing rough concepts. Formally, from three mutually disjoint sets $\mathsf{N_C}$, $\mathsf{N_R}$, and $\mathsf{N_I}$ of *concept*, *role*, and *individual* names, *rough* $\mathcal{EL}^{++}$ *concepts* are constructed using the following syntactic rule:

$$C \quad ::= \quad A \quad | \quad C_1 \sqcap C_2 \quad | \quad \exists r.C \quad | \quad \overline{C} \quad | \quad \underline{C} \quad | \quad \{a\} \quad | \quad \top \quad | \quad \bot,$$

where $A \in \mathsf{N_C}$, $r \in \mathsf{N_R}$, and $a \in \mathsf{N_I}$.[4]

The semantics of this logic is based on interpretations that map concept names to subsets of a non-empty domain Δ, and role names to binary relations over Δ. To handle the rough concept constructors, these interpretations additionally require an indiscernibility relation.

Definition 1. *A* rough interpretation *is a tuple $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}, \sim_{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty set called the* domain, *$\sim_{\mathcal{I}}$ is an equivalence relation on $\Delta^{\mathcal{I}}$, called the* indiscernibility relation, *and $\cdot^{\mathcal{I}}$ is the* interpretation function *mapping every concept name A to a subset $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, every role name r to a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and every individual name a to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$.*

As usual, we denote the equivalence class of an element $x \in \Delta^{\mathcal{I}}$ w.r.t. the relation $\sim_{\mathcal{I}}$ by $[x]_{\sim_{\mathcal{I}}}$. The interpretation function is extended to general rough $\mathcal{EL}^{++}$ concepts by setting:

- $(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$,
- $(\exists r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y \in \Delta^{\mathcal{I}}.\ (x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$,
- $\overline{C}^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid [x]_{\sim_{\mathcal{I}}} \cap C^{\mathcal{I}} \neq \emptyset\}$,
- $\underline{C}^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid [x]_{\sim_{\mathcal{I}}} \subseteq C^{\mathcal{I}}\}$,
- $\{a\}^{\mathcal{I}} = \{a^{\mathcal{I}}\}$,
- $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$, and $\bot^{\mathcal{I}} = \emptyset$.

Intuitively, the indiscernibility relation groups the elements of the domain that cannot be distinguished from each other, at the considered level of detail. The upper approximation $\overline{C}$ of a given concept C describes those individuals that cannot be excluded from belonging to C, as they are indistinguishable from some element belonging to this concept. Dually, the individuals $\underline{C}$ are those that are discernible (i.e., can be detached) from every element *not* belonging to C. Clearly, for every interpretation $\mathcal{I}$ and concept C it holds that $\underline{C}^{\mathcal{I}} \subseteq C^{\mathcal{I}} \subseteq \overline{C}^{\mathcal{I}}$. The borderline cases, those elements belonging to $\overline{C}^{\mathcal{I}} \setminus \underline{C}^{\mathcal{I}}$, cannot be ensured to be, nor excluded from being instances of C through the equivalence relation.

The domain knowledge is described using a *TBox*: a finite set of *GCIs* of the form $C \sqsubseteq D$, where C, D are rough $\mathcal{EL}^{++}$ concepts, and *role inclusion axioms* (RIs) of the form $r \circ s \sqsubseteq t$ or $r \sqsubseteq t$, where $r, s, t \in \mathsf{N_R}$. Their semantics is defined as follows. The interpretation $\mathcal{I}$ *satisfies* the GCI $C \sqsubseteq D$ if and only if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds. It satisfies the RI $r \circ s \sqsubseteq t$ (resp., $r \sqsubseteq t$) if $r^{\mathcal{I}} \circ s^{\mathcal{I}} \subseteq t^{\mathcal{I}}$ (resp., $r^{\mathcal{I}} \subseteq t^{\mathcal{I}}$). $\mathcal{I}$ is a *model* of the TBox $\mathcal{T}$ if it satisfies all the GCIs and RIs in $\mathcal{T}$.

Contrary to less expressive DLs such as $\mathcal{EL}$, it is possible to build inconsistent rough $\mathcal{EL}^{++}$ TBoxes, due to the presence of the bottom concept $\bot$. As a simple example, consider the GCI $\{a\} \sqsubseteq \bot$ that cannot be satisfied by any interpretation $\mathcal{I}$. Despite this situation, we still focus our attention to the problem of deciding subsumption between concepts, which can be used to solve all

[4] The logic $\mathcal{EL}^{++}$ allows also for concrete domains. In this paper we decided to exclude concrete domains to reduce the number of completion rules, and simplify the proofs. Including this constructor in the logic should not affect our complexity results.

other standard reasoning problems like concept satisfiability, or the instance problem [1].

Definition 2. *Let $\mathcal{T}$ be a TBox and C, D two rough $\mathcal{EL}^{++}$ concepts. We say that C is* subsumed *by D w.r.t. $\mathcal{T}$, denoted by $C \sqsubseteq_\mathcal{T} D$, if for every model $\mathcal{I}$ of $\mathcal{T}$ it holds that $C^\mathcal{I} \subseteq D^\mathcal{I}$.* Classification *is the problem of deciding, for every pair of concept names A, B, whether $A \sqsubseteq_\mathcal{T} B$ holds or not.*

Example 3. Consider once again the Ensatina salamanders and the TBox

$$\begin{aligned} \mathsf{MontereyE} \sqcap \mathsf{LargeBlotchedE} &\sqsubseteq \bot \\ \exists \mathsf{interbreed}.\mathsf{MontereyE} &\sqsubseteq \overline{\mathsf{MontereyE}} \\ \exists \mathsf{interbreed}.\mathsf{LargeBlotchedE} &\sqsubseteq \overline{\mathsf{LargeBlotchedE}} \end{aligned}$$

that describes usual desired properties of the notion of species; namely, that no individual may belong to two different species (first axiom), and that belonging to a species is characterized by the capacity of interbreeding with elements of that species (last two axioms). Consider now three salamanders a, b, c such that a and c belong to each of the limit species, and b can interbreed with both; i.e.,

$$\begin{aligned} \{a\} &\sqsubseteq \mathsf{MontereyE} \\ \{c\} &\sqsubseteq \mathsf{LargeBlotchedE} \\ \{b\} &\sqsubseteq \exists \mathsf{interbreed}.\{\mathsf{a}\} \sqcap \exists \mathsf{interbreed}.\{\mathsf{c}\}. \end{aligned}$$

From all these axioms, we can deduce that the salamander b belongs to the upper approximation of both limit species, and hence is an intermediate salamander. We could then deduce some further properties of $\{b\}$ in the presence of other axioms in the TBox.

If we had restricted the description to the classical definition of species through interbreeding, i.e., used the axioms $\exists \mathsf{interbreed}.\mathsf{MontereyE} \sqsubseteq \mathsf{MontereyE}$ and $\exists \mathsf{interbreed}.\mathsf{LargeBlotchedE} \sqsubseteq \mathsf{LargeBlotchedE}$ in place of the upper approximations as above, the TBox would be inconsistent as b would be a member of both species, which are specified to be disjoint. In this case, rough concepts provide a (partial) solution to the species problem.

As shown in [22], reasoning in rough DLs can be reduced to reasoning in a classical DL that allows value restrictions, inverse, and reflexive roles, and role inclusion axioms. Let ρ be a new role that does not appear in $\mathcal{T}$. If we restrict ρ to be reflexive, and include the role inclusion axioms $\rho \circ \rho \sqsubseteq \rho$ (transitivity), and $\rho^{-1} \sqsubseteq \rho$ (symmetry), then the concepts $\overline{C}$ and $\underline{C}$ are equivalent to the concepts $\exists \rho.C$ and $\forall \rho.C$, respectively (see [22] for full details). However, although transitive roles are a feature of $\mathcal{EL}^{++}$, it is well known that extensions of classical $\mathcal{EL}^{++}$ with either value restrictions or inverse roles are already intractable; in fact reasoning in these extensions is ExpTime-complete [1,2,24]. Applying this reduction directly, yields an ExpTime upper bound for the complexity of deciding subsumption of rough $\mathcal{EL}^{++}$ concepts. On the other hand, only one role name, namely ρ, is used in any of the possibly expensive constructors introduced by this reduction. As we will see in the following section, this limited use does help in improving the complexity, as the problem of deciding subsumption between concepts is decidable in polynomial time.

Clearly, the subsumption relation $\sqsubseteq_\mathcal{T}$ is transitive; that is, if $C \sqsubseteq_\mathcal{T} D$ and $D \sqsubseteq_\mathcal{T} E$, then also $C \sqsubseteq_\mathcal{T} E$ holds. Due to the properties of lower and upper approximations, some additional subsumption relations can sometimes be deduced, as shown next.

Theorem 4. *For all rough $\mathcal{EL}^{++}$ concepts C, D, E, D_1, D_2, the following properties hold:*

1. $\overline{C} \sqsubseteq_\mathcal{T} D$ *iff* $C \sqsubseteq_\mathcal{T} \underline{D}$
2. *if* $C \sqsubseteq_\mathcal{T} \overline{D}$ *and* $D \sqsubseteq_\mathcal{T} \underline{E}$*, then* $C \sqsubseteq_\mathcal{T} \underline{E}$
3. *if* $C \sqsubseteq_\mathcal{T} \underline{D}$ *and* $\underline{D} \sqsubseteq_\mathcal{T} E$*, then* $C \sqsubseteq_\mathcal{T} \underline{E}$
4. *if* $C \sqsubseteq_\mathcal{T} \underline{D_1}$ *and* $C \sqsubseteq_\mathcal{T} \underline{D_2}$ *(respectively,* $C \sqsubseteq_\mathcal{T} D_2$*, or* $C \sqsubseteq_\mathcal{T} \overline{D_2}$*), then* $C \sqsubseteq_\mathcal{T} \underline{D_1 \sqcap D_2}$ *(resp.,* $C \sqsubseteq_\mathcal{T} D_1 \sqcap D_2$*, or* $C \sqsubseteq_\mathcal{T} \overline{D_1 \sqcap D_2}$*).*

Proof. Let $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I}, \sim_\mathcal{I})$ be a model of $\mathcal{T}$, and $x \in \Delta^\mathcal{I}$.

1. ($\Leftarrow$) If $x \in \overline{C}^\mathcal{I}$, then there exists a $y \in [x]_{\sim_\mathcal{I}} \cap C^\mathcal{I}$. By assumption, $y \in \underline{D}^\mathcal{I}$. Thus, $x \in [y]_{\sim_\mathcal{I}} \subseteq D^\mathcal{I}$.

   ($\Rightarrow$) Let $x \in C^\mathcal{I}$. We must prove that $[x]_{\sim_\mathcal{I}} \subseteq D^\mathcal{I}$. Let $y \sim_\mathcal{I} x$. Then, $y \in \overline{C}^\mathcal{I}$, and thus, by assumption, $y \in D^\mathcal{I}$.
2. Let $x \in C^\mathcal{I}$. By assumption, we know that there exists $z \in [x]_{\sim_\mathcal{I}} \cap D^\mathcal{I}$, and thus $z \in \underline{E}^\mathcal{I}$; i.e., $[x]_{\sim_\mathcal{I}} = [z]_{\sim_\mathcal{I}} \subseteq E^\mathcal{I}$. Hence $x \in \underline{E}^\mathcal{I}$.
3. If $x \in C^\mathcal{I}$, then by assumption it holds that $[x]_{\sim_\mathcal{I}} \subseteq D^\mathcal{I}$. Let $y \sim_\mathcal{I} x$. Then $[y]_{\sim_\mathcal{I}} = [x]_{\sim_\mathcal{I}} \subseteq D^\mathcal{I}$, and hence $y \in \underline{D}^\mathcal{I}$, and by assumption $y \in E^\mathcal{I}$.
4. If $x \in C^\mathcal{I}$, then $[x]_{\sim_\mathcal{I}} \subseteq D_1^\mathcal{I}$. For the case where $C \sqsubseteq_\mathcal{T} \underline{D_2}$, it then follows that $[x]_{\sim_\mathcal{I}} \subseteq D_1^\mathcal{I} \cap D_2^\mathcal{I} = (D_1 \sqcap D_2)^\mathcal{I}$, and hence $x \in (\underline{D_1 \sqcap D_2})^\mathcal{I}$. If $C \sqsubseteq_\mathcal{T} D_2$, then $x \in D_2^\mathcal{I}$, and since $x \in [x]_{\sim_\mathcal{I}}$, it follows that $x \in (D_1 \sqcap D_2)^\mathcal{I}$. Finally, if $C \sqsubseteq_\mathcal{T} \overline{D_2}$, then $[x]_{\sim_\mathcal{I}} \cap D_2^\mathcal{I} \neq \emptyset$ and since $[x]_{\sim_\mathcal{I}} \subseteq D_1^\mathcal{I}$, it holds that $[x]_{\sim_\mathcal{I}} \cap D_1^\mathcal{I} = [x]_{\sim_\mathcal{I}}$. Thus, $[x]_{\sim_\mathcal{I}} \cap (D_1 \sqcap D_2)^\mathcal{I} \neq \emptyset$. □

In the following section we will exploit these properties to build a completion-based algorithm that classifies a TBox and can be used to decide which subsumption relations hold.

4 A Completion Algorithm

In this section, we describe an algorithm for deciding subsumption relations between concepts. To simplify the description, we will focus exclusively on subsumption between concept *names*. Notice that subsumption between complex rough $\mathcal{EL}^{++}$ concepts C, D can be reduced to this problem by adding the two axioms $A \sqsubseteq C$ and $D \sqsubseteq B$, where A, B are two new concept names, to $\mathcal{T}$ and then deciding whether $A \sqsubseteq_\mathcal{T} B$ holds. Thus, restricting to concept name subsumption results in no loss of generality.

Table 1. Normalisation rules, where $A \in \mathsf{BC}, C, D \notin \mathsf{BC}$ and X is a new concept name

NF1	$A \sqcap C \sqsubseteq E$	$\longrightarrow$	$\{C \sqsubseteq X, A \sqcap X \sqsubseteq E\}$
NF2	$\exists r.C \sqsubseteq E$	$\longrightarrow$	$\{C \sqsubseteq X, \exists r.X \sqsubseteq E\}$
NF3	$\underline{C} \sqsubseteq E$	$\longrightarrow$	$\{C \sqsubseteq X, \underline{X} \sqsubseteq E\}$
NF4	$\overline{C} \sqsubseteq E$	$\longrightarrow$	$\{C \sqsubseteq \underline{E}\}$
NF5	$C \sqsubseteq D$	$\longrightarrow$	$\{C \sqsubseteq X, X \sqsubseteq D\}$
NF6	$A \sqsubseteq E \sqcap F$	$\longrightarrow$	$\{A \sqsubseteq E, A \sqsubseteq F\}$
NF7	$A \sqsubseteq \exists r.C$	$\longrightarrow$	$\{A \sqsubseteq \exists r.X, X \sqsubseteq C\}$
NF8	$A \sqsubseteq \underline{C}$	$\longrightarrow$	$\{A \sqsubseteq \underline{X}, X \sqsubseteq C\}$
NF9	$A \sqsubseteq \overline{C}$	$\longrightarrow$	$\{A \sqsubseteq \overline{X}, X \sqsubseteq C\}$
NF10	$\bot \sqsubseteq E$	$\longrightarrow$	$\emptyset$

As a preprocessing step for the algorithm, we transform the TBox into an adequate normal form. We define the set BC of *basic concepts* as the smallest set containing all concept names, all nominal concepts, and the top concept; i.e., $\mathsf{BC} := \mathsf{N_C} \cup \{\top\} \cup \{\{a\} \mid a \in \mathsf{N_I}\}$. The TBox $\mathcal{T}$ is in *normal form*, if all its GCIs are of one of the following forms:

$$A \sqsubseteq \exists r.B, \quad \exists r.A \sqsubseteq C, \quad A \sqcap A' \sqsubseteq C, \quad \underline{A} \sqsubseteq C, \quad A \sqsubseteq \underline{B}, \ \text{or} \quad A \sqsubseteq \overline{B},$$[5]

where $A, A', B \in \mathsf{BC}$, $C \in \mathsf{BC} \cup \{\bot\}$, and $r \in \mathsf{N_R}$. The normalisation rules shown in Table 1 can be used to transform any TBox $\mathcal{T}$ into a TBox in normal form that preserves all the subsumption relations from $\mathcal{T}$. It is possible to show that these normalisation rules yield a normalised TBox in linear time. Notice in particular rule **NF4**, which takes advantage of the first property described in Theorem 4.

Our completion algorithm extends the methods described in [1], to appropriately handle the lower and upper approximations of concepts. The idea is to store the information of the subsumption relations using a collection of *completion sets*. The main difference with the classical approach is that we need to maintain special completion sets for the lower and upper approximations, in order to handle the special properties of these constructors. Moreover, as shown in [15], a correct handling of nominals requires to keep track of additional dependencies between basic concepts. This is done through a reachability relation $\rightsquigarrow_R$, where $A \rightsquigarrow_R B$ intuitively expresses that if A has a non-empty interpretation, then B must also be non-empty. This relation is built in parallel to the completion sets during the execution of the algorithm.

The algorithm uses a family of completion sets as data structure. In the following we will denote as $\mathsf{BC}_\mathcal{T}$ the set of all basic concepts that appear in the TBox $\mathcal{T}$, and analogously for $\mathsf{N_C}_\mathcal{T}$, $\mathsf{N_R}_\mathcal{T}$, and $\mathsf{N_I}_\mathcal{T}$. For every basic concept $A \in \mathsf{BC}_\mathcal{T}$ and every concept name $G \in \mathsf{N_C}_\mathcal{T}$, we store three completion sets $S^G(A), \underline{S}^G(A)$, and $\overline{S}^G(A)$, and additionally a completion set $S^G(A, r)$ for every role name $r \in \mathsf{N_R}_\mathcal{T}$.

[5] To simplify the description, we use the expression $\top \sqcap A \sqsubseteq B$ to represent axioms of the form $A \sqsubseteq B$.

The members of the completion sets are all basic concepts or $\bot$. These sets will maintain the following invariants during the whole execution of the algorithm:

I1 if $B \in S^G(A)$, and $G \rightsquigarrow_R A$, then $A \sqsubseteq_\mathcal{T} B$
I2 if $B \in \overline{S}^G(A)$, and $G \rightsquigarrow_R A$, then $A \sqsubseteq_\mathcal{T} \overline{B}$
I3 if $B \in \underline{S}^G(A)$, and $G \rightsquigarrow_R A$, then $A \sqsubseteq_\mathcal{T} \underline{B}$
I4 if $B \in S^G(A, r)$, and $G \rightsquigarrow_R A$, then $A \sqsubseteq_\mathcal{T} \exists r.B$
I5 if $G \rightsquigarrow_R A$, then for every model $\mathcal{I}$ of $\mathcal{T}$, $G^\mathcal{I} \neq \emptyset$ implies $A^\mathcal{I} \neq \emptyset$.

The completion sets are initialised as

$$S^G(A) = \overline{S}^G(A) := \{A, \top\}, \quad \underline{S}^G(A) := \{\top\}, \quad S^G(A, r) := \emptyset$$

for basic concepts $A \in \mathsf{BC}_\mathcal{T}$, concept names $G \in \mathsf{N_C}_\mathcal{T}$, and role names $r \in \mathsf{N_R}_\mathcal{T}$. The reachability relation initially states only that $G \rightsquigarrow_R G$ and $G \rightsquigarrow_R \{a\}$ for every $G \in \mathsf{N_C}_\mathcal{T}$ and every $a \in \mathsf{N_I}_\mathcal{T}$. Obviously, this initialisation preserves all the invariants described above.

The completion rules from Table 2 are then applied to extend these sets. Before continuing to show correctness of this algorithm, we briefly explain these rules. The rules up to CR7 correspond to the completion rules for classical $\mathcal{EL}^{++}$ from [1] with the correct treatment of nominals adapted from [15]. The following rules up to CR15 consider the axioms containing rough concepts, as well as the consequences of crisp axioms when applied to rough concepts. The first two of those rules are a simple consequence of the properties of intersections of rough sets. We discuss the rule CR12 in more detail. Under the assumption that G is not empty, $G \rightsquigarrow_R A_2$ states that A_2 must also be non-empty. Additionally, $\{a\} \in \underline{S}^G(A_2)$ in particular implies that every member of A_2 must also belong to $\{a\}$, and hence A_2 must be equivalent to $\{a\}$. Consider now some element of A_1. $\{a\} \in \overline{S}^G(A_1)$ states that this element must be indiscernible from a and hence is indiscernible from an element of A_2. Thus, A_2 must be added to $\underline{S}^G(A_1)$. The rule CR11 follows from a similar but simpler argument.

The next six rules consider a cross-population of of the completion sets, following the properties of rough sets described in the previous section. Finally, the last two rules extend the reachability relation to keep information on which concept names should be interpreted as non-empty under the assumption that G is non-empty.

To ensure termination, a rule is only applied if it adds new information; that is, if the basic concepts to be added to the completion sets by such rule application are not already in them. These rules are applied until the completion sets are *saturated*; i.e., until no rule is applicable anymore. We first show that this procedure terminates in polynomial time.

Lemma 5. *The rules from Table 2 can only be applied a polynomial number of times, and each rule application needs polynomial time.*

Table 2. Completion rules for rough $\mathcal{EL}^{++}$

CR1	if $B_1 \in S^G(A), B_2 \in S^G(A)$, and $B_1 \sqcap B_2 \sqsubseteq C \in \mathcal{T}$, then add C to $S^G(A)$
CR2	if $B \in S^G(A)$ and $B \sqsubseteq \exists r.C \in \mathcal{T}$, then add C to $S^G(A, r)$
CR3	if $B \in S^G(A, r), C \in S^G(B)$, and $\exists r.C \sqsubseteq D \in \mathcal{T}$, then add D to $S^G(A)$
CR4	if $B \in S^G(A, r)$ and $\bot \in S^G(B)$, then add $\bot$ to $S^G(A)$
CR5	if $B \in S^G(A, r)$ and $r \sqsubseteq t \in \mathcal{T}$, then add B to $S^G(A, t)$
CR6	if $B \in S^G(A, r), C \in S^G(B, s)$, and $r \circ s \sqsubseteq t \in \mathcal{T}$, then add C to $S^G(A, t)$
CR7	if $\{a\} \in S^G(A_1) \cap S^G(A_2)$ and $G \rightsquigarrow_R A_2$, then add A_2 to $S^G(A_1)$
CR8	if $B_1 \in \underline{S}^G(A), B_2 \in \underline{S}^G(A)$, and $B_1 \sqcap B_2 \sqsubseteq C \in \mathcal{T}$, then add C to $\underline{S}^G(A)$
CR9	if $B_1 \in \underline{S}^G(A), B_2 \in \overline{S}^G(A)$, and $B_1 \sqcap B_2 \sqsubseteq C \in \mathcal{T}$, then add C to $\overline{S}^G(A)$
CR10	if $B \in \overline{S}^G(A)$ and $\bot \in \overline{S}^G(B)$, then add $\bot$ to $\underline{S}^G(A)$
CR11	if $\{a\} \in \overline{S}^G(A_1) \cap \overline{S}^G(A_2)$ and $G \rightsquigarrow_R A_2$, then add A_2 to $\overline{S}^G(A_1)$
CR12	if $\{a\} \in \overline{S}^G(A_1) \cap \underline{S}^G(A_2)$ and $G \rightsquigarrow_R A_2$, then add A_2 to $\underline{S}^G(A_1)$
CR13	if $B \in \underline{S}^G(A)$ and $\underline{B} \sqsubseteq C \in \mathcal{T}$, then add C to $\underline{S}^G(A)$
CR14	if $B \in \overline{S}^G(A)$, and $B \sqsubseteq \underline{C} \in \mathcal{T}$, then add C to $\underline{S}^G(A)$
CR15	if $B \in \overline{S}^G(A)$, and $B \sqsubseteq \overline{C} \in \mathcal{T}$, then add C to $\overline{S}^G(A)$
CR16	if $B \in \underline{S}^G(A)$ then add B to $S^G(A)$
CR17	if $B \in S^G(A)$ then add B to $\overline{S}^G(A)$
CR18	if $B \in \underline{S}^G(A)$ and $C \in S^G(B)$ then add C to $\underline{S}^G(A)$
CR19	if $B \in \overline{S}^G(A)$ and $C \in \overline{S}^G(B)$ then add C to $\overline{S}^G(A)$
CR20	if $B \in \overline{S}^G(A)$ and $C \in \underline{S}^G(B)$ then add C to $\underline{S}^G(A)$
CR21	if $\bot \in \overline{S}^G(A)$ then add $\bot$ to $\underline{S}^G(A)$
CR22	if $G \rightsquigarrow_R A$ and $B \in S^G(A, r)$, then $G \rightsquigarrow_R B$
CR23	if $G \rightsquigarrow_R A$ and $B \in \overline{S}^G(A)$, then $G \rightsquigarrow_R B$

Proof. Each of the completion sets contains only basic concepts that appear in $\mathcal{T}$. Thus, the size of each of these sets is linear on $\mathcal{T}$. For each concept name in $\mathcal{T}$ there are three such completion sets for every basic concept, plus one additional completion set for each basic concept and role name. Thus, the number of completion sets is quadratic on the size of $\mathcal{T}$. Each application of a completion rule CR1–CR21 adds one concept name to one completion set, and never removes any. This means that there can be at most polynomially many rule applications, before no new concept name can be added to any completion set. The reachability relation $\rightsquigarrow_R$ maps basic concepts, so it can have at most quadratically many elements. Each application of one of the last two rules adds a pair to this relation, and hence only quadratically many rule applications are possible.

For testing the pre-condition of a rule application, we can simply explore all the completion sets, at most twice, and the set of axioms $\mathcal{T}$. This exploration needs in total polynomial time. □

When the algorithm terminates, we can read all the subsumption relations between concept names appearing in the TBox $\mathcal{T}$, by simply considering the elements appearing in the subsumption sets. More precisely, the subsumption relation $A \sqsubseteq_\mathcal{T} B$ holds iff (i) $\{B, \bot\} \cap S^A(A) \neq \emptyset$, or (ii) there exists $a \in \mathsf{N}_{\mathsf{I}\mathcal{T}}$ such that $\bot \in S^A(\{a\})$. We prove first that the method is sound, by showing that rule applications preserve the invariants i1 to i5 described before.

Lemma 6. *The invariants* i1 *to* i5 *are preserved through all rule applications.*

Proof. As said before, the invariants are satisfied by the initialisation of the completion sets. Soundness of the first seven rules has been shown in [1,15]. For the remaining rules, we take advantage of the properties of rough concepts. Recall that for every concept name A, it holds that $\underline{A} \sqsubseteq_\mathcal{T} A \sqsubseteq_\mathcal{T} \overline{A}$. This shows soundness of the rules cr16 and cr17.

For the rule cr8, let $A \sqsubseteq_\mathcal{T} \underline{B_1}$ and $A \sqsubseteq_\mathcal{T} \underline{B_2}$. Then for every interpretation $\mathcal{I}$ and every $x \in \mathcal{I}$ if $x \in A^\mathcal{I}$, then $[x]_{\sim_\mathcal{I}} \subseteq B_1^\mathcal{I} \cap B_2^\mathcal{I}$. Thus, $[x]_{\sim_\mathcal{I}} \subseteq C^\mathcal{I}$, which implies that $A \sqsubseteq \underline{C}$. Rule cr9 can be treated analogously. Soundness of the rules treating nominals has been argued before, and of the remaining concept rules is a direct consequence of Theorem 4.

The last two rules simply transfer the assumption of non-emptiness to all existential successors, in the first case, and to all weak subsumers in the second case. This transfer preserves the invariant i5. □

Since $A \rightsquigarrow_R A$, the first invariant entails that whenever $B \in S^A(A)$, the subsumption relation $A \sqsubseteq_\mathcal{T} B$ holds. Likewise, if $\bot \in S^A(A) \cup S^A(\{a\})$, the same invariant together with i5 yield that A must be interpreted as empty by every model of $\mathcal{T}$. If this is the case, then A is trivially subsumed by B.

It remains only to show completeness; i.e., that once the algorithm has terminated, all the subsumption relations are explicitly stated in the completion sets, as described before. As usual, we show this by building, given concept names $A, B \in \mathsf{N}_{\mathsf{C}\mathcal{T}}$ not satisfying the conditions (i) nor (ii) above, a countermodel for the subsumption relation between A and B. The main idea is to have one domain element for each basic concept C appearing in $\mathcal{T}$, which can be reached from A through the relation $\rightsquigarrow_R$ (and thus, must have a non-empty interpretation in the countermodel). The interpretation function will include this element in every basic concept D that subsumes C w.r.t. $\mathcal{T}$. However, we need to create additional auxiliary individuals to correctly deal with the upper and lower approximations of each of these concept names. We thus add an element C_u that will be interpreted to belong to all concept names D such that $\underline{D}$ subsumes C. For dealing with the upper approximations, the construction is slightly more complex, as different elements might be needed to witness the existence of an indiscernible element belonging to different concept names: from $C \sqsubseteq_{\mathcal{T}} \overline{D_1}$ and $C \sqsubseteq_{\mathcal{T}} \overline{D_2}$, and $x \in C^{\mathcal{I}}$, we can only deduce that there exist y_1 and y_2 such that $x{\sim_{\mathcal{I}}}y_i$ and $y_i \in D_i^{\mathcal{I}}$ holds for $i \in \{1, 2\}$. If we enforce $y_1 = y_2$, then it would follow that $x \in (\overline{D_1 \sqcap D_2})^{\mathcal{I}}$, but this is *not* a consequence of the two subsumptions. Thus, we need to treat the witnesses for C being subsumed by $\overline{D_1}$ and by $\overline{D_2}$ independently. Moreover, since nominals must be interpreted as singleton sets, we also need to identify all basic concepts that are subsumed by the same nominal. We also need to identify the auxiliary domain elements introduced for dealing with rough constructors, if they refer to the same nominal, or if they were generated by a conjunction of lower and upper approximations. We formalize these ideas next.

Lemma 7. *Let A, B be two concept names appearing in $\mathcal{T}$, and S^A the class completion sets for A obtained after the application of the completion rules has terminated. If $\{B, \bot\} \cap S^A(A) = \emptyset$ and $\bot \notin S^A(\{a\})$, for all $a \in \mathsf{N_I}_{\mathcal{T}}$, then $A \not\sqsubseteq_{\mathcal{T}} B$.*

Proof. We need to build a model $\mathcal{I}$ of $\mathcal{T}$ such that $A^{\mathcal{I}} \not\subseteq B^{\mathcal{I}}$. We start by defining the set of relevant concepts

$$\mathcal{C} := \{C, C_u, C_D \mid C, D \in \mathsf{BC}_{\mathcal{T}}, A{\rightsquigarrow_R}C, D \in \overline{S}^A(C)\}.$$

Let $\bowtie$ be the relation on $\mathcal{C}$ where $x \bowtie y$ iff any of the following conditions hold:

1. exist $a \in \mathsf{N_I}_{\mathcal{T}}$, $C, D \in \mathsf{BC}_{\mathcal{T}}$ with $x = C$, $y = D$, and $\{a\} \in S^A(C) \cap S^A(D)$,
2. exist $a \in \mathsf{N_I}_{\mathcal{T}}$ and $C \in \mathsf{BC}_{\mathcal{T}}$ with $x = \{a\}$ and $y = C_{\{a\}}$,
3. exists $C \in \mathsf{BC}_{\mathcal{T}}$ with $x = C, y = C_u$ and $C \in \underline{S}^A(C)$, or
4. exist $C, D_1, D_2, E \in \mathsf{BC}_{\mathcal{T}}$ with $x = C_{D_1}$ $y = C_E$, $D_2 \in \underline{S}^A(C), D_1 \in \overline{S}^A(C)$, and $D_1 \sqcap D_2 \sqsubseteq E \in \mathcal{T}$.

Let $\ulcorner C \urcorner$ denote the equivalence class of C on the transitive, reflexive and symmetric closure of $\bowtie$. These equivalence classes form the interpretation domain; that is, $\Delta^{\mathcal{I}} := \{\ulcorner C \urcorner \mid C \in \mathcal{C}\}$.

The idea is to use the class $\ulcorner C \urcorner$ as a prototype individual belonging to the concept C. Recall that we have assumed that $\bot \notin S^A(A) \cup \bigcup_{a \in \mathsf{N_I}_{\mathcal{T}}} S^A(\{a\})$. From this assumption it follows that $\bot$ does not appear in any equivalence class.

The indiscernibility relation $\sim_{\mathcal{I}}$ is the transitive, reflexive and symmetric closure of $\{(\ulcorner C \urcorner, \ulcorner C_u \urcorner), (\ulcorner C \urcorner, \ulcorner C_D \urcorner) \mid C, D \in \mathsf{BC}_{\mathcal{T}}, A{\rightsquigarrow_R}C\}$; thus, the indiscernibility class defined by a basic concept C is

$$[C]_{\sim_{\mathcal{I}}} := \{\ulcorner C \urcorner, \ulcorner C_u \urcorner\} \cup \{\ulcorner C_D \urcorner \mid D \in \overline{S}^A(C)\}.$$

It remains only to define the interpretation function $\cdot^{\mathcal{I}}$. For a concept name $C \in \mathsf{N_C}_{\mathcal{T}}$, role name $r \in \mathsf{N_R}_{\mathcal{T}}$ and individual name $a \in \mathsf{N_I}_{\mathcal{T}}$, we set

$$\begin{aligned}
C^{\mathcal{I}} :=\ & \{\ulcorner D \urcorner \mid C \in S^A(D)\} \cup \{\ulcorner D_u \urcorner \mid C \in \underline{S}^A(D)\} \cup \\
& \{\ulcorner D_X \urcorner \mid C \in S^A(X), X \in \overline{S}^A(D)\} \cup \{\ulcorner D_X \urcorner \mid C \in \underline{S}^A(D), D_X \in \mathcal{C}\}, \\
r^{\mathcal{I}} :=\ & \{(\ulcorner C \urcorner, \ulcorner D \urcorner) \mid D \in S(C,r)\} \cup \{(\ulcorner C_u \urcorner, \ulcorner D \urcorner) \mid D \in S(X,r), X \in \underline{S}(C)\} \cup \\
& \{(\ulcorner C_X \urcorner, \ulcorner D \urcorner) \mid D \in S(X,r), X \in \overline{S}(C)\} \cup \\
& \{(\ulcorner C_X \urcorner, \ulcorner D \urcorner) \mid D \in S(Y,r), Y \in \underline{S}(C), C_X \in \mathcal{C}\}, \text{ and} \\
a^{\mathcal{I}} :=\ & \ulcorner \{a\} \urcorner.
\end{aligned}$$

It can be seen that this interpretation function is well defined. It is a simple case analysis to show that, for every $C \in \mathsf{BC}_{\mathcal{T}}$, and every $D \in \mathsf{BC}_{\mathcal{T}} \cup \{\bot\}$ it holds that $\ulcorner C \urcorner \in D^{\mathcal{I}}$ iff $D \in S^A(C)$. Hence, we have that $\ulcorner A \urcorner \in A^{\mathcal{I}}$ but $\ulcorner A \urcorner \notin B^{\mathcal{I}}$. It only remains to be shown that $\mathcal{I}$ is indeed a model of $\mathcal{T}$. The proof is by case analysis, on the shape of the axiom, and the domain element.

[$\underline{C} \sqsubseteq D$] Let $x \in \underline{C}^{\mathcal{I}}$; i.e., $[x]_{\sim_{\mathcal{I}}} \subseteq C^{\mathcal{I}}$ and let $E \in \mathsf{BC}_{\mathcal{T}}$ such that $[E]_{\sim_{\mathcal{I}}} = [x]_{\sim_{\mathcal{I}}}$. Then $\ulcorner E_u \urcorner \in C^{\mathcal{I}}$. By definition, this means that $C \in \underline{S}(E)$. Since the rule CR13 is not applicable, $D \in \underline{S}(E)$, and by rule CR16, $D \in S(E)$. Let now $\ulcorner E_F \urcorner \in [E]_{\sim_{\mathcal{I}}}$. Since $D \in \underline{S}(E)$, by definition $\ulcorner E_F \urcorner \in D^{\mathcal{I}}$. It thus follows that $[E]_{\sim_{\mathcal{I}}} \subseteq D^{\mathcal{I}}$ and since $x \in [E]_{\sim_{\mathcal{I}}}$, $x \in D^{\mathcal{I}}$.

[$C \sqsubseteq \underline{D}$] Let $x \in C^{\mathcal{I}}$ and $E \in \mathsf{BC}_{\mathcal{T}}$ with $[x]_{\sim_{\mathcal{I}}} = [E]_{\sim_{\mathcal{I}}}$. Then, x is one of $\ulcorner E_u \urcorner, \ulcorner E \urcorner$, or $\ulcorner E_F \urcorner$ for some $F \in \mathsf{BC}_{\mathcal{T}}$. Since $x \in C^{\mathcal{I}}$, by definition we know that either $C \in \underline{S}^A(E)$, $C \in S^A(E)$, or $C \in S^A(F)$ and $F \in \overline{S}^A(E)$, depending on the shape of x. In any of the three cases, saturation of the rules CR16, CR17, and CR19, implies that $C \in \overline{S}(E)$. By rule CR14, it the follows that $D \in \underline{S}(E)$ and hence also $D \in S(E) \cap \overline{S}(E)$. This implies that $[x]_{\sim_{\mathcal{I}}} = [E]_{\sim_{\mathcal{I}}} \subseteq D^{\mathcal{I}}$, and thus $x \in \underline{D}^{\mathcal{I}}$.

[$C \sqsubseteq \overline{D}$] Let $x \in C^{\mathcal{I}}$ and $[x]_{\sim_{\mathcal{I}}} = [E]_{\sim_{\mathcal{I}}}$. As in the previous case, we know that $C \in \overline{S}(E)$, and from rule CR15 it follows that $D \in \overline{S}(E)$. Thus, $\ulcorner E_D \urcorner \in D^{\mathcal{I}}$. Since $\ulcorner E_D \urcorner \in [x]_{\sim_{\mathcal{I}}}$, this implies that $[x]_{\sim_{\mathcal{I}}} \cap D^{\mathcal{I}} \neq \emptyset$, and hence $x \in \overline{D}^{\mathcal{I}}$.

[$C_1 \sqcap C_2 \sqsubseteq D$] Let $x \in C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$ and $E \in \mathsf{BC}_{\mathcal{T}}$ such that $[x]_{\sim_{\mathcal{I}}} = [E]_{\sim_{\mathcal{I}}}$. If $x = \ulcorner E \urcorner$, then by definition $\{C_1, C_2\} \subseteq S^A(E)$, and from rule CR1 it follows that $D \in S^A(E)$ and hence $x = \ulcorner E \urcorner \in D^{\mathcal{I}}$. The case for $x = \ulcorner E_u \urcorner$ can be shown analogously using rule CR8. If $x = \ulcorner E_F \urcorner$ for some $F \in \overline{S}^A(E)$, then for each $i \in \{1, 2\}$ it holds that $C_i \in S^A(F) \cup \underline{S}^A(E)$. The cases where both C_is belong to the same set are analogous to the cases for $\ulcorner E \urcorner$ and $\ulcorner E_u \urcorner$ shown before. For the remaining two cases assume w.l.o.g. that $C_1 \in S^A(F)$ and $C_2 \in \underline{S}^A(E)$. Then, by rule CR17 $C_1 \in \overline{S}^A(F)$. The definition of the relation $\bowtie$ then implies that $\ulcorner E_F \urcorner = \ulcorner E_D \urcorner$. From rules CR19 and CR9 it also follows that $D \in \overline{S}^A(E)$. These two facts together imply that $x = \ulcorner E_D \urcorner \in D^{\mathcal{I}}$.

The remaining cases can be treated in a similar way, following the arguments for the classical setting from [1,15]. The only additional difficulty arises in a case analysis for the shape of the domain elements, as the classes for C_u and C_D depend on the completion sets $\underline{S}$ and $\overline{S}$, which have a slightly different behaviour than S. □

This lemma shows that the algorithm is complete. In order to decide whether a concept name A is subsumed by $B \in \mathsf{N}_{\mathsf{C}\mathcal{T}}$, one needs only analyse the sets $S^A(A)$ and $S^A(\{a\})$ for all $a \in \mathsf{N}_{\mathsf{I}\mathcal{T}}$. If the goal is to classify the TBox $\mathcal{T}$, then this analysis has to be repeated for all concept names A, however, there is no need to recompute the completion sets; one run of the completion algorithm provides information on all the subsumption relations between concept names. We thus obtain the following result.

Theorem 8. *Subsumption of rough $\mathcal{EL}^{++}$ concept names w.r.t. TBoxes can be decided in polynomial time. Moreover, the TBox $\mathcal{T}$ can be classified in polynomial time.*

Since subsumption is already PTime-hard for classical $\mathcal{EL}$ [12], this theorem proves that the problem is PTime-complete.

5 Conclusions

We have studied rough $\mathcal{EL}^{++}$, a description logic that extends the lightweight DL $\mathcal{EL}^{++}$ to allow for lower and upper approximations from rough set theory. Rough DLs are presented as an alternative to fuzzy DLs for dealing with imprecise knowledge, in face to the recent negative complexity results for fuzzy description logics. Rough DLs allow for a less fine-grained treatment of vagueness, which reflects in a lower complexity of reasoning.

The logic we studied covers the logical basis for the OWL 2 EL profile of the standard ontology language for the semantic web OWL 2, except for the expression of concrete domains. We have shown that subsumption of concept names w.r.t. rough $\mathcal{EL}^{++}$ TBoxes can be decided in polynomial time. This result was obtained by providing a completion-based algorithm capable of classifying the TBox in polynomial time. As an added benefit, our approach does not require including expensive constructors that damage the efficiency of $\mathcal{EL}^{++}$ reasoners. We do not expect that adding p-admissible concrete domains to this formalism would negatively affect these complexity results.

Our algorithm is a direct extension from the one presented in [1] in that, when no rough constructors appear in the TBox, the algorithm behaves similarly. The only difference is in the handling of nominals, where we adapt the method from [15] to obtain completeness. Unfortunately, the cost of handling potential rough concepts is to double the space needed.[6] This unnecessary cost can be easily avoided by disallowing applications of rules CR8 to CR21 and rule CR23 whenever the TBox uses only classical $\mathcal{EL}^{++}$ constructors. Our algorithm requires maintaining a higher number of completion sets and dealing with a

[6] Without the lower approximation constructor, the sets $\underline{S}$ are never populated.

larger variety of rules. Despite this, the structure of these completion sets and rules is very similar to the ones used in current implementations of $\mathcal{EL}^{++}$ reasoners. Thus, we do not expect that implementing them into a rough $\mathcal{EL}^{++}$ system would cause much trouble.

These polynomial-time complexity results give strength to the observation from [22] that rough constructors can be added to classical DLs with no additional cost in terms of complexity.

We should emphasize that in this paper we have considered only classical subsumption in a rough description logic. There exist other non-standard reasoning services that consider rough concepts in higher detail, as described in [17]. As presented in this paper, our completion algorithm is incapable of solving those reasoning tasks.

As part of our future work, we intend to study the complexity of rough-set-specific reasoning problems for rough $\mathcal{EL}^{++}$ and, if possible, extend our completion algorithm to handle them adequately. We also intend to extend our algorithm to deal with concrete domains, hence covering the whole OWL 2 EL profile. Finally, we intend to implement the system and use it for applications that require the representation of imprecise knowledge.

References

1. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: Proc. 19th Int. Joint Conf. on Artif. Intel. (IJCAI 2005), Edinburgh, UK. Morgan-Kaufmann Publishers (2005)
2. Baader, F., Brandt, S., Lutz, C.: Pushing the el envelope further. In: Clark, K., Patel-Schneider, P.F. (eds.) Proceedings of the OWLED 2008 DC Workshop on OWL: Experiences and Directions (2008)
3. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications, 2nd edn. Cambridge University Press (2007)
4. Bobillo, F., Straccia, U.: Finite fuzzy description logics and crisp representations. In: Bobillo, F., Costa, P.C.G., d'Amato, C., Fanizzi, N., Laskey, K.B., Laskey, K.J., Lukasiewicz, T., Nickles, M., Pool, M. (eds.) URSW 2008-2010/UniDL 2010. LNCS (LNAI), vol. 7123, pp. 99–118. Springer, Heidelberg (2013)
5. Borgwardt, S., Distel, F., Peñaloza, R.: How fuzzy is my fuzzy description logic? In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 82–96. Springer, Heidelberg (2012)
6. Borgwardt, S., Peñaloza, R.: Undecidability of fuzzy description logics. In: Brewka, G., Eiter, T., McIlraith, S.A. (eds.) Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012), Rome, Italy, pp. 232–242. AAAI Press (2012)
7. Borgwardt, S., Peñaloza, R.: The complexity of lattice-based fuzzy description logics. Journal on Data Semantics 2(1), 1–19 (2013)
8. Borgwardt, S., Peñaloza, B.: Positive subsumption in fuzzy $\mathcal{EL}$ with general t-norms. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013), Beijing, China. AAAI Press (to appear, 2013)
9. Cerami, M., Straccia, U.: On the (un)decidability of fuzzy description logics under Łukasiewicz t-norm. Information Sciences 227, 1–21 (2013)

10. d'Amato, C., Fanizzi, N., Esposito, F., Lukasiewicz, T.: Representing uncertain concepts in rough description logics via contextual indiscernibility relations. In: Bobillo, F., et al. (eds.) URSW 2008-2010/UniDL 2010. LNCS (LNAI), vol. 7123, pp. 300–314. Springer, Heidelberg (2013)
11. de Queiroz, K.: Different species problems and their resolution. BioEssays 27(12), 1263–1269 (2005)
12. Haase, C.: Complexity of subsumption in extensions of $\mathcal{EL}$. Master's thesis, Dresden University of Technology, Germany (2007)
13. Hájek, P.: Metamathematics of Fuzzy Logic (Trends in Logic). Springer (2001)
14. Jiang, Y., Wang, J., Tang, S., Xiao, B.: Reasoning with rough description logics: An approximate concepts approach. Information Sciences 179(5), 600–612 (2009)
15. Kazakov, Y., Kroetzsch, M., Simancik, F.: Practical reasoning with nominals in the el family of description logics. In: Brewka, G., Eiter, T., McIlraith, S.A. (eds.) Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012), pp. 264–274. AAAI Press (2012)
16. Kazakov, Y., Krötzsch, M., Simančík, F.: ELK reasoner: Architecture and evaluation. In: Proceedings of the OWL Reasoner Evaluation Workshop 2012 (ORE 2012). CEUR Workshop Proceedings, vol. 858. CEUR-WS.org (2012)
17. Maria Keet, C.: Rough subsumption reasoning with rowl. In: Brown, I., Sewchurran, K., Suleman, H. (eds.) Proc. of the 2011 Annual Conf. of the South African Inst. of Comp. Scientists and Inform. Tech. (SAICSIT 2011), pp. 133–140. ACM (2011)
18. Liau, C.-J.: On rough terminological logics. In: Proc. of the 4th Intern. Workshop on Rough Sets, Fuzzy Sets and Machine Discovery, pp. 47–54 (1996)
19. Lukasiewicz, T., Straccia, U.: Managing uncertainty and vagueness in description logics for the semantic web. Journal of Web Semantics 6(4), 291–308 (2008)
20. Pawlak, Z.: Rough sets. International Journal of Parallel Programming 11(5), 341–356 (1982)
21. Peñaloza, R., Zou, T.: Rough $\mathcal{EL}$ classification. In: Proceedings of the 2013 International Workshop on Description Logics (DL 2013). CEUR-WS, Ulm, Germany (to appear, 2013)
22. Schlobach, S., Klein, M.C.A., Peelen, L.: Description logics with approximate definitions - precise modeling of vague concepts. In: Veloso, M.M. (ed.) Proc. 20th Int. Joint Conf. on Artif. Intel. (IJCAI 2007), pp. 557–562 (2007)
23. Straccia, U.: Reasoning within fuzzy description logics. J. Artif. Intell. Res. 14, 137–166 (2001)
24. Toman, D., Weddell, G.: On reasoning about structural equality in xml: a description logic approach. Theor. Comput. Sci. 336(1), 181–203 (2005)
25. Yao, Y.: Perspectives of granular computing. In: Proceeding of the 2005 IEEE International Conference on Granular Computing, vol. 1, pp. 85–90 (2005)
26. Zadeh, L.A.: Fuzzy sets. Information and Control 8(3), 338–353 (1965)

Uniform Interpolation of $\mathcal{ALC}$-Ontologies Using Fixpoints

Patrick Koopmann* and Renate A. Schmidt

The University of Manchester, UK
{koopmanp,schmidt}@cs.man.ac.uk

Abstract. We present a method to compute uniform interpolants with fixpoints for ontologies specified in the description logic $\mathcal{ALC}$. The aim of uniform interpolation is to reformulate an ontology such that it only uses a specified set of symbols, while preserving consequences that involve these symbols. It is known that in $\mathcal{ALC}$ uniform interpolants cannot always be finitely represented. Our method computes uniform interpolants for the target language $\mathcal{ALC}\mu$, which is $\mathcal{ALC}$ enriched with fixpoint operators, and always computes a finite representation. If the result does not involve fixpoint operators, it is the uniform interpolant in $\mathcal{ALC}$. The method focuses on eliminating concept symbols and combines resolution-based reasoning with an approach known from the area of second-order quantifier elimination to introduce fixpoint operators when needed. If fixpoint operators are not desired, it is possible to approximate the interpolant.

1 Introduction

Ontologies represent information about concepts and relations (roles) using description logics, fragments of first-order logic, to allow reasoning systems to derive implicit information automatically. There are situations where it is useful to restrict an ontology to a subset of the vocabulary without affecting the meaning of the remaining concepts. When reusing parts from a general ontology for a specific domain, this can be done by restricting the ontology to the concepts that are known and interesting in this domain. Instead of restricting an ontology to a more specific domain, another application is restricting the ontology to a set of higher level concepts to create a summary of the ontology. Another example is hiding confidential concepts, which is useful when an ontology is shared or published, but some information should be kept secret [9].

In uniform interpolation, the ontology is reformulated in such a way that only symbols from a specified set are used, while logical consequences over the remaining symbols are preserved [4]. This paper describes a method for uniform interpolation of ontologies represented in the description logic $\mathcal{ALC}$.

Uniform interpolation for $\mathcal{ALC}$ is not a new topic. In [18], a method based on tableaux reasoning was published. In [12] theoretical properties of uniform

* Patrick Koopmann is supported by an EPSRC EU Doctoral Training Award.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 87–102, 2013.

interpolation were presented, among them that uniform interpolants can in the worst case be of size triple exponential in the size of the original ontology.

A problem of uniform interpolation in $\mathcal{ALC}$ is that the interpolants cannot always be represented using a finite number of finite $\mathcal{ALC}$-axioms. We offer a solution to this problem by using $\mathcal{ALC}\mu$, which is $\mathcal{ALC}$ enriched with fixpoint operators [11], to represent interpolants. This way we show how to always compute a finitely represented interpolant. Since fixpoint operators are not common in the description logic community yet, we also describe ways of approximating the result, in case no adequate representation without fixpoints is possible.

Our method combines two approaches known from the context of second-order quantifier elimination [7]. First, we use a resolution-based calculus to eliminate symbols in a focused way. Resolution-based methods have been used for eliminating symbols in different logics, like first order logic [6] or modal logic [10], but these logics are not expressive enough if the result requires fixpoint operators. We use a clausal form based on structural transformation, where new concept symbols are introduced dynamically, in order to deal with this. Afterwards we use a variation of the generalised Ackermann's Lemma [14] to eliminate these introduced symbols and add fixpoint operators when necessary.

2 Preliminaries

Let N_c, N_r be two disjoint sets of *concept symbols* and *role symbols*. *Concepts* in $\mathcal{ALC}$ are of the following form:

$$\bot \mid \top \mid A \mid \neg C \mid C \sqcup D \mid C \sqcap D \mid \exists r.C \mid \forall r.C,$$

where $A \in N_c$, $r \in N_r$ and C and D are arbitrary concepts. $\top$, $C \sqcap D$ and $\forall r.C$ are defined as abbreviations: $\top$ stands for $\neg\bot$, $C \sqcap D$ for $\neg(\neg C \sqcup \neg D)$ and $\forall r.C$ for $\neg\exists r.\neg C$.

A TBox is a set of *axioms* of the forms $C \sqsubseteq D$ and $C \equiv D$, where C and D are concepts. $C \equiv D$ is a short-hand for the two axioms $C \sqsubseteq D$ and $D \sqsubseteq C$. Since we are only dealing with the TBox part of an ontology, we will use the terms 'ontology' and 'TBox' interchangeably.

The semantics of $\mathcal{ALC}$ is defined as follows. An *interpretation* is a pair $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$, where the *domain* $\Delta^{\mathcal{I}}$ is a nonempty set and the *interpretation function* $\cdot^{\mathcal{I}}$ assigns to each concept symbol $A \in N_c$ a subset of $\Delta^{\mathcal{I}}$ and to each role symbol $r \in N_r$ a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation function is extended to concepts as follows:

$$\bot^{\mathcal{I}} := \emptyset \qquad (\neg C)^{\mathcal{I}} := \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \qquad (C \sqcup D)^{\mathcal{I}} := C^{\mathcal{I}} \cup D^{\mathcal{I}}$$
$$(\exists r.C)^{\mathcal{I}} := \{x \in \Delta^{\mathcal{I}} \mid \exists y : (x,y) \in r^{\mathcal{I}} \land y \in C^{\mathcal{I}}\}.$$

$C \sqsubseteq D$ is *true* in an interpretation $\mathcal{I}$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. $\mathcal{I}$ is model of a TBox $\mathcal{T}$ if all axioms in $\mathcal{T}$ are true in $\mathcal{I}$. A TBox $\mathcal{T}$ is *satisfiable* if there exists a model for $\mathcal{T}$, otherwise it is *unsatisfiable.* $\mathcal{T} \models C \sqsubseteq D$ holds iff in every model of $\mathcal{T}$ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Two TBoxes $\mathcal{T}_1$ and $\mathcal{T}_2$ are *equi-satisfiable* if every model of $\mathcal{T}_1$ can be extended to a model of $\mathcal{T}_2$, and vice versa.

In order to define $\mathcal{ALC}\mu$, we extend the language with a set N_v of *concept variables.* $\mathcal{ALC}\mu$ extends $\mathcal{ALC}$ with concepts of the form $\mu X.C$ and $\nu X.C$, where $X \in N_v$, and C is a concept in which X occurs as a concept symbol only positively (under an even number of negations). $\mu X.C$ is the *least fixpoint* of C on X and $\nu X.C$ the *greatest fixpoint.*

A concept variable X is *bound* if it occurs in the scope C of a fixpoint expression $\mu X.C$ or $\nu X.C$. Otherwise it is *free.* A concept is *closed* if it does not contain any free variables. Axioms in $\mathcal{ALC}\mu$ are of the form $C \sqsubseteq D$ and $C \equiv D$, where C and D are closed concepts.

Following [3], we define the semantics of fixpoint expressions. Let $\mathcal{V}$ be an *assignment function* that maps concept variables to subsets of $\Delta^{\mathcal{I}}$. $\mathcal{V}[X \mapsto W]$ denotes $\mathcal{V}$ modified by setting $\mathcal{V}(X) = W$. $C^{\mathcal{I},\mathcal{V}}$ is the interpretation of C taking into account this assignment, and when $\mathcal{V}$ is defined for all variables in C, $C^{\mathcal{I},\mathcal{V}} = C^{\mathcal{I}}$. The semantics of fixpoint concepts is defined as follows:

$$(\mu X.C)^{\mathcal{I},\mathcal{V}} := \bigcap\{W \subseteq \Delta^{\mathcal{I}} \mid C^{\mathcal{I},\mathcal{V}[X\mapsto W]} \subseteq W\}$$
$$(\nu X.C)^{\mathcal{I},\mathcal{V}} := \bigcup\{W \subseteq \Delta^{\mathcal{I}} \mid W \subseteq C^{\mathcal{I},\mathcal{V}[X\mapsto W]}\}.$$

3 Overview of the Method

We are interested in computing *uniform interpolants* of TBoxes. Let *sig*(C) denote the set of concept symbols occurring in the concept C and *sig*$(\mathcal{T})$ the set of concept symbols occurring in the TBox $\mathcal{T}$.

Definition 1. *Given a TBox $\mathcal{T}$ and a set Σ of concept symbols, the TBox $\mathcal{T}'$ is a* uniform interpolant *of $\mathcal{T}$ over Σ iff (i) sig$(\mathcal{T}') \subseteq \Sigma$, and (ii) for every $C \sqsubseteq D$ with sig$(C \sqcap D) \subseteq \Sigma$: $\mathcal{T}' \models C \sqsubseteq D$ iff $\mathcal{T} \models C \sqsubseteq D$.*

Observe that from this definition follows that uniform interpolants of a TBox over a given set of concept symbols are unique modulo logical equivalence. Figure 1 gives an outline of our method for computing uniform interpolants. In Phase 1, the ontology is transformed into a set of clauses. In Phase 2, we process each concept symbol A that occurs in the TBox but not in the set Σ one after another. We saturate the set of clauses with respect to A using a set of rules and eliminate clauses containing A. This is described in more detail in Section 4. Both phases may introduce new symbols, which are eliminated Phase 3, which is described in more detail in Section 5. This phase may involve the use of fixpoint operators, if these symbols are cyclic. After this phase, the uniform interpolant is already computed, but we add a fourth phase that applies simplifications and converts resulting axioms to proper subsumption relations.

The order in which symbols are processed in Phase 2 and 3 is not crucial, since we can prove that the result is always the correct uniform interpolant. Different orders of symbol elimination may lead to different syntactic representations, but these are logically equivalent. The following theorem states the correctness of our method and is proven in Section 7.

1. Compute the clausal representation $N := \mathit{clauses}(\mathcal{T})$ of $\mathcal{T}$.
2. For every $A \in \mathit{sig}(\mathcal{T}) \setminus \Sigma$:
   - eliminate A by setting $N := \mathit{ELIM}_{\mathit{Res}}(N, A)$. $\mathit{ELIM}_{\mathit{Res}}$ uses a resolution based procedure described in Section 4. This step introduces definer symbols.
3. Set $\mathcal{T} = N$. For every definer D in the resulting clause set:
   - eliminate D by setting $\mathcal{T} := \mathit{ELIM}_{\mathit{Ack}}(\mathcal{T}, D)$. $\mathit{ELIM}_{\mathit{Ack}}$ uses Ackermann's Lemma and may introduce fixpoints. This is described in Section 5.
4. Apply further simplifications to $\mathcal{T}$ if possible and return the resulting ontology.

Fig. 1. The complete method for computing uniform interpolants

Theorem 1. *For any $\mathcal{ALC}$ TBox $\mathcal{T}$ and any signature Σ, our method terminates and returns a finite representation of the uniform interpolant of $\mathcal{T}$ over Σ in the description logic $\mathcal{ALC}\mu$. If the result does not make use of the greatest fixpoint operator, the result is the uniform interpolant of $\mathcal{T}$ over Σ in $\mathcal{ALC}$.*

4 Resolution-Based Symbol Elimination

In this section we describe the first two phases of our method in more detail, that is, transformation to clausal form and elimination of symbols. The latter is based on a new resolution calculus for deciding satisfiability in $\mathcal{ALC}$, which we describe as well.

Let $N_D \subseteq N_c$ be a set of designated concept symbols called *definers*, which do not occur in the input ontology.

Definition 2. *An $\mathcal{ALC}$-*literal *is a concept description of the form A, $\neg A$, $\forall r.D$ or $\exists r.D$, where A is a concept symbol, r is a role symbol and D is a definer.*

*A TBox is in $\mathcal{ALC}$-*conjunctive normal form *if every axiom is of the form $\top \sqsubseteq L_1 \sqcup ... \sqcup L_n$, where each L_i is an $\mathcal{ALC}$-literal. The right part of such a subsumption is called $\mathcal{ALC}$-*clause*. In the following we assume $\mathcal{ALC}$-clauses are represented as sets of literals (this means no clause contains the same literal more than once). The empty clause is denoted by $\bot$ and represents a contradiction.*

Every $\mathcal{ALC}$ TBox can be transformed into an equi-satisfiable TBox in $\mathcal{ALC}$-conjunctive normal form using structural transformation. This can be achieved by first transforming the input TBox into negation normal form, incrementally replacing every concept C that occurs immediately below a role restriction by a definer D and adding the axiom $D \sqsubseteq C$ for each such subconcept. The resulting TBox does not contain any nested role restrictions and can be brought into $\mathcal{ALC}$-conjunctive normal form by applying standard CNF-transformation techniques. It is crucial for our method of computing uniform interpolants that the structural transformation is performed in this way.

For a TBox $\mathcal{T}$, let $\mathit{clauses}(\mathcal{T})$ refer to the set of clauses generated in this way. The set $\mathit{clauses}(\mathcal{T})$ is produced by Phase 1.

Resolution:

$$\frac{C_1 \sqcup A \qquad C_2 \sqcup \neg A}{C_1 \sqcup C_2}$$

provided $C_1 \cup C_2$ does not contain more than one negative definer-literal.

Role Propagation:

$$\frac{C_1 \sqcup \forall r.D_1 \qquad C_2 \sqcup Qr.D_2}{C_1 \sqcup C_2 \sqcup Qr.D_3}$$

where $Q \in \{\exists, \forall\}$ and D_3 is a (possibly new) definer representing $D_1 \sqcap D_2$, provided $C_1 \cup C_2$ does not contain more than one negative definer-literal.

Existential Role Restriction Elimination:

$$\frac{C \sqcup \exists R.D \qquad \neg D}{C}$$

Fig. 2. Rules of decision procedure $RES_{\mathcal{ALC}}$

Example 1. Consider the following TBox $\mathcal{T}$:

$$A \sqsubseteq B \sqcup C \qquad B \sqsubseteq \exists r.B \qquad C \sqsubseteq \forall r.\neg B$$

The obtained clause set *clauses*($\mathcal{T}$) is the following:

1. $\neg A \sqcup B \sqcup C$ 3. $\neg D_1 \sqcup B$ 5. $\neg D_2 \sqcup \neg B$
2. $\neg B \sqcup \exists r.D_1$ 4. $\neg C \sqcup \forall r.D_2$

where D_1 and D_2 are definers introduced during the structural transformation.

The resolution method used to eliminate symbols in Phase 2 is based on a new resolution-based decision procedure, $RES_{\mathcal{ALC}}$, which we describe next. $RES_{\mathcal{ALC}}$ uses the rules shown in Figure 2. The *resolution rule* is a variation of the classical resolution rule for propositional logic. Its side condition ensures that every derived clause contains at most one negative definer literal, a property needed for the successful elimination of the introduced definer symbols in Phase 3. Another motivation for this side condition is that clauses with different negative definer literals represent concepts occurring under different role restrictions. A combination of these only makes sense if the contexts of these role restrictions get combined as well. This is performed by the *role propagation rule*: it propagates the conceptual information under a universal role restriction into concepts occurring under other role restrictions, and creates a clause that represents the combined contexts of the definers. The role propagation rule is based on the logical entailment $\models ((A \sqcup B) \sqcap (C \sqcup D)) \sqsubseteq (A \sqcup C \sqcup (B \sqcap D))$.

Since the normal form has to be preserved, the role propagation rule may require the introduction of a new definer symbol D_3 representing the conjunction of the definers D_1 and D_2 occurring in the premises. This is done by adding new clauses $\neg D_3 \sqcup D_1$ and $\neg D_3 \sqcup D_2$ to the clause set. We refer to this as *combining D_1 and D_2 into a new definer D_3*. Observe that the resolution rule also applies to definer-literals. This way for each pair of clauses $\neg D_1 \sqcup C_1$ and $\neg D_2 \sqcup C_2$ we derive the clauses $\neg D_3 \sqcup C_1$ and $\neg D_3 \sqcup C_2$, for which the side conditions of the rules are satisfied.

In order to avoid the infinite introduction of new definers, we keep track of introduced definers and reuse them when possible. Let N_D* denote the definer symbols that were introduced by the initial normal form transformation in Phase 1, which we call *base definers*. The mapping $\mathit{conj} : N_D \mapsto 2^{N_D*}$ maps each definer to the set of base definers it represents. If a definer representing $D_1 \sqcap D_2$ is needed, we check whether the mapping already maps a definer to $\mathit{conj}(D_1) \cup \mathit{conj}(D_2)$. If it does, we reuse it; if not, we add a new definer together with the required axioms. This way the number of introduced symbols is bounded by $2^{|N_D*|}$.

It is not hard to see that the rules in Figure 2 are sound. This means saturating a set of clauses always produces an equi-satisfiable set of clauses. The existential role restriction elimination rule is sound, because the clause $\neg D$ represents the axiom $D \sqsubseteq \bot$. Since the introduction of definers preserves equi-satisfiability, we can state soundness of the calculus:

Lemma 1. *The calculus $RES_{\mathcal{ALC}}$ is sound, that is, for any TBox $\mathcal{T}$, the saturation of clauses($\mathcal{T}$) using the rules of $RES_{\mathcal{ALC}}$ is equi-satisfiable with $\mathcal{T}$, and if the empty clause can be derived, $\mathcal{T}$ is unsatisfiable.*

We can also prove refutational completeness and termination. The proof is given Section 6.

Theorem 2. *$RES_{\mathcal{ALC}}$ is sound and refutationally complete, and provides a decision procedure for TBox satisfiability in $\mathcal{ALC}$.*

This result is used in Section 7 to prove the correctness of our method to compute uniform interpolants.

The rules of $RES_{\mathcal{ALC}}$ are used in Phase 2 for saturating and eliminating symbols. In particular, in order to eliminate a concept symbol A from a set N of $\mathcal{ALC}$-clauses, we restrict the rules to be applied only on the literals $\neg A$, A, $\neg D$, D, $\forall r.D$ and $\exists r.D$, where A is the symbol we want to eliminate and D is a definer connected to A. A definer D is *connected to a concept symbol A* if D either co-occurs with A in a clause or if D co-occurs in a clause with another definer D' that is connected to A. After N is saturated using these restricted rules, we remove all clauses containing A and all clauses of the form $\neg D \sqcup D'$, where D and D' is any definer, since they are not required anymore. We call this method $\mathit{ELIM}_{\mathit{Res}}$ and denote the resulting set of clauses by $\mathit{ELIM}_{\mathit{Res}}(N, A)$.

Theorem 3. *Given the clausal representation N of a TBox $\mathcal{T}$ and a concept symbol A, $\mathit{ELIM}_{\mathit{Res}}(N, A)$ is computed in finitely bounded time, does not contain A and preserves all consequences over $\Sigma = \mathit{sig}(\mathcal{T}) \setminus \{A\}$.*

It is worth mentioning that both $RES_{\mathcal{ALC}}$ and $ELIM_{Res}$ can make use of standard redundancy elimination techniques used in resolution-based theorem proving, including tautology and subsumption deletion, which we omit here for space reasons.

Example 2. We demonstrate the application of $RES_{\mathcal{ALC}}$ on the clause set generated in the last example. Suppose we want to compute the uniform interpolant over $\Sigma = \{A, C\}$, which means B is the only concept symbol we have to eliminate. Resolution on the B-literals in clauses 3 and 5 would produce a clause with two different negative definer-literals, thus violating the side condition of the resolution rule (see Figure 2). But we can combine the definers D_1 and D_2 by applying the role propagation rule. Note that both are connected to B, and that the role propagation rule is applicable to clauses 2 and 4. This leads to the introduction of the definer D_3 representing $D_1 \sqcap D_2$, and the clauses capturing $D_3 \sqsubseteq D_1 \sqcap D_2$.

6. $\neg B \sqcup \neg C \sqcup \exists r.D_3$ (*role prop. between 2 and 4*)
7. $\neg D_3 \sqcup D_1$ $(D_3 \sqsubseteq D_1)$
8. $\neg D_3 \sqcup D_2$ $(D_3 \sqsubseteq D_2)$

Observe that an additional application of the role propagation rule on the same clauses does not result in new clauses, since a definer representing $D_1 \sqcap D_2$ has already been introduced. The new clauses 7 and 8 can now be resolved on the positive definer literals.

9. $\neg D_3 \sqcup B$ (*resolution between 3 and 7*)
10. $\neg D_3 \sqcup \neg B$ (*resolution between 5 and 8*)

Now we have two clauses that allow resolving on B, resulting in a clause that makes the existential role restriction elimination rule applicable:

11. $\neg D_3$ (*resolution between 9 and 10*)
12. $\neg B \sqcup \neg C$ (*exist. elim. between 6 and 11*)

The last clause expresses the disjointness of the concepts B and C, which is a consequence of the last two axioms of the sample TBox. Further applications of the resolution rule are possible, which we omit for space reasons.

5 Eliminating Definers Using Ackermann's Lemma

In Phase 3, the definers that have been introduced in Phase 2 are eliminated. This may involve the introduction of fixpoint operators. We also describe how to approximate the uniform interpolant in $\mathcal{ALC}$.

The main idea of this phase is captured in the following theorem:

Theorem 4. *Let $\mathcal{T}$ be a TBox which contains an axiom of the form $A \sqsubseteq C$, where A is a concept symbol that occurs only positively in the rest of $\mathcal{T}$.*

(i) If C does not contain A, the uniform interpolant of $\mathcal{T}$ over $sig(\mathcal{T}) \setminus \{A\}$ is obtained by removing that axiom and replacing every other occurrence of A in the rest of the ontology by C.

(ii) If C contains A positively, the interpolant is obtained by removing that axiom and replacing every occurrence of A with $\nu X.C'$, where C' is acquired from C by replacing every A with the fresh concept variable X.

This theorem is a translation of Ackermann's Lemma, which was first published in [1] and generalised for the fixpoint case in [14], to description logic syntax. Ackermann's Lemma and its generalisation have been used in the context of second-order quantifier elimination to eliminate existentially quantified predicate variables in second-order logic expressions [14,7].

The underlying idea of the theorem is that if there is a definition of A in the ontology, we can use this definition to replace all occurrences of A in order to eliminate A. If the definition is cyclic, we have to use a fixpoint operator.

We use this theorem in Phase 3 to eliminate the introduced definer symbols. The method to compute $ELIM_{Ack}(\mathcal{T}, D)$ consists of the following steps:

1. Group all axioms of the form $\top \sqsubseteq \neg D \sqcup C_i$ into a single axiom of the form $D \sqsubseteq C_D$, where $C_D = \bigsqcap_i C_i$. If there is no such clause, set $C_D = \top$.
2. Remove the axiom $D \sqsubseteq C_D$ from $\mathcal{T}$.
3. If D does not occur in C_D, replace every occurrence of D in $\mathcal{T}$ with C_D.
4. If D occurs in C_D, replace every occurrence of D in $\mathcal{T}$ with $\nu X.C'_D$, where C'_D is acquired from C_D by replacing D with X, where X is a fresh concept variable not used in $\mathcal{T}$.

If the output of the algorithm contains fixpoints there are two ways in which we can approximate the result in $\mathcal{ALC}$: signature approximation and semantic approximation. In *signature approximation*, we return a finite TBox equi-satisfiable with the uniform interpolant which approximates the signature Σ and therefore may contain additional concept symbols. This is done by not eliminating definers which would lead to the use of fixpoint operators.

In contrast, using *semantic approximation*, we return a result that is completely in the specified signature Σ, but approximates the interpolant semantically. For this, we omit Step 3 and apply Step 2 above incrementally for a specified number of times even if D occurs in C_D, and replace it afterwards by $\top$. This way the semantics of the greatest fixpoint operator is approximated in the result. This solution is similar to the one offered in [18].

Example 3. We continue on the last example. The result of $ELIM_{Res}(N, B)$ is equivalent to the following ontology.[1]

1. $\top \sqsubseteq \neg A \sqcup C \sqcup \exists r.D_1$
2. $\top \sqsubseteq \neg C \sqcup \forall r.D_2$
3. $D_1 \sqsubseteq \neg C \sqcap \exists r.D_1$
4. $D_2 \sqsubseteq \neg A \sqcup C$
5. $D_3 \sqsubseteq \bot$

[1] To simplify the example, we left out redundant and tautological clauses, which would otherwise be removed in Phase 4 by the described method.

Axiom 5 can be ignored since D_3 does not occur in the rest of the ontology. D_2 can be eliminated by replacing it with $\neg A \sqcup C$. The elimination of D_1 leads to the introduction of a fixpoint operator. After applying simplifications (Phase 4), we obtain the following ontology, which is the uniform interpolant of our sample TBox for $\Sigma = \{A, C\}$:

$$6.\ A \sqsubseteq C \sqcup \exists r.\nu X.(\neg C \sqcap \exists r.X) \qquad\qquad 7.\ C \sqsubseteq \forall r.(\neg A \sqcup C)$$

We cannot express this uniform interpolant in a finite way in $\mathcal{ALC}$, but we can approximate it signature-wise and semantically. The signature approximation is acquired by not eliminating D_1 and including Axiom 3 in the result. For the semantic approximation, we would replace D_1 n times by $C \sqcap \exists r.D_1$, and then replace it by $\top$. For $n = 2$, Axiom 6 would be approximated as follows:

$$6'.\ A \sqsubseteq C \sqcup \exists r.(\neg C \sqcap \exists r.(\neg C \sqcap \exists r.\top)).$$

6 Correctness of the Decision Procedure

In this section we prove termination and refutational completeness of $RES_{\mathcal{ALC}}$. This result is needed in the next section to prove the correctness of $ELIM_{Res}$.

Lemma 2. *$RES_{\mathcal{ALC}}$ always terminates and produces at most $2^{2n_c+(2+2n_r)\cdot 2^{n_d}}$ clauses, where n_c is the number of concept symbols in the input ontology, n_r the number of role symbols and n_d the number of base definers introduced by the normal form transformation.*

Proof. Because of how we keep track of newly introduced definers, we have maximally 2^{n_d} many definers in the result. Each definer D can occur in the forms D, $\neg D$, $\exists r.D$ and $\forall r.D$, and each concept symbol A can occur in a positive or a negative literal, which means there are $2n_c + (2 + 2n_r) \cdot 2^{n_d}$ many possible literals. Every literal can only occur once in a clause (clauses are represented as sets), which gives us the worst case upper bound of $2^{2n_c+(2+2n_r)\cdot 2^{n_d}}$. □

In order to prove completeness of $RES_{\mathcal{ALC}}$, we use a candidate model construction approach similar to the one used to prove refutational completeness for ordered resolution [2] and refutational completeness for consequence-driven reasoners for description logics [16]. We show that for each set of clauses saturated using the rules of our calculus and not containing the empty clause, we can construct a candidate model which is actually a model for the set.

The construction is done in the following way. For each satisfiable definer concept D, we create a set I^D of literals that have to be satisfied by a domain element in order to satisfy the definer. A special definer ϵ is used to represent concepts that do not occur under a role restriction. We then create a domain element x_D for each definer D and construct an interpretation in such a way that every atomic concept and every existential restriction in each I^D is satisfied. We show that the resulting interpretation is indeed a model for our saturated set of clauses.

Let N_s denote a set of clauses saturated using the rules of $RES_{\mathcal{ALC}}$. The set $\mathbf{D}$ consists of all definers used in N_s and the special symbol ϵ. N_s is partitioned into a set of *definition sets*: the function $d : \mathbf{D} \longrightarrow 2^{N_s}$ maps each definer $D \in \mathbf{D}$ to the subset of clauses in N_s which have $\neg D$ as a literal, and ϵ to all remaining clauses. Because of the side conditions of the rules (every derived clause can have at most one negative definer-literal), such a partitioning is always possible. $d(D)$ contains all clauses that make up the *definition* of D, in the sense that they can be represented in an axiom of the form $D \sqsubseteq ...$, hence we use the terminology *definition set* for $d(D)$. $d(\epsilon)$ contains all the remaining clauses, which are not related to the definition of any definer. Let $d^e(D) = d(D) \cup d(\epsilon)$ be the definition set extended with these clauses. If a domain element satisfies D, it also has to satisfy all clauses in $d^e(D)$, and it suffices to check the clauses in $d^e(D)$ to check whether an instance satisfies D or not.

We define a partial ordering $\sqsubseteq_D$ on definers in the following way: $D_1 \sqsubseteq_D D_2$ iff $conj(D_2) \subseteq conj(D_1)$ (see Section 4 for the definition of *conj*). This ordering represents the subsumption hierarchy between the respective conjunctions of base-definers, because $\models \bigsqcap conj(D_1) \sqsubseteq \bigsqcap conj(D_2)$ if $conj(D_2) \subseteq conj(D_1)$.

We define an ordering $\prec_L$ on literals that satisfies the following constraints:

- $D \prec_L \neg D \prec_L A \prec_L \neg A \prec_L \exists r.D' \prec_L \forall r.D''$ for all atomic concepts A that are not definers, for all roles r and for all definers D, D', D''.
- If $D_1 \sqsubseteq_D D_2$, then $D_1 \prec_L D_2$, $\exists r.D_1 \prec_L \exists r.D_2$ and $\forall r.D_1 \prec_L \forall r.D_2$ for all roles r. (From this follows that if D represents $D_1 \sqcap D_2$, then $\exists r.D \prec_L \exists r.D_1$ and $\forall r.D \prec_L \forall r.D_2$.)

It can be shown that an ordering with these constraints always exists. $\prec_L$ is extended to an ordering $\prec_C$ between clauses using the multiset-extension $(\prec_L)_{mul}$ of $\prec_L$. Using $\prec_C$, the clauses in each $d^e(D)$ are enumerated: C_i^D denotes the ith clause in $d^e(D)$ according to $\prec_C$, starting from the smallest clause.

Following this enumeration, we define a set I^D of positive literals for each element $D \in \mathbf{D}$ (including ϵ), such that if a domain element x satisfies every literal in I^D, it also satisfies $d^e(D)$. For a set of positive literals I, we say *I satisfies a literal L, taking into account the subsumption hierarchy on $\mathbf{D}$*, written $I \models_{\mathbf{D}} L$, iff (i) L is a positive literal of the form A and $A \in I$, (ii) L is a negative literal of the form $\neg A$ and $A \notin I$, (iii) L is of the form $\exists r.D$ and there is a $\exists r.D' \in I$ with $D' \sqsubseteq_D D$ or (iv) L is of the form $\forall r.D$ and for every literal of the form $\exists r.D' \in I$ we have $D' \sqsubseteq_D D$. We say *I satisfies a clause C*, written $I \models_{\mathbf{D}} C$, if there is a literal $L \in C$ such that $I \models_{\mathbf{D}} L$.

We define I^D formally in five steps:

1. If $\neg D \in d(D)$, set $I^D = \emptyset$. Otherwise, let
2. $I_0^D = \{D\}$ if $D \neq \epsilon$ and $I_0^D = \emptyset$ if $D = \epsilon$.
3. $I_i^D = I_{i-1}^D \cup \{L\}$, if $I_{i-1}^D \not\models_{\mathbf{D}} C_{i-1}^D$ and the maximal literal L of C_{i-1}^D is a positive literal of the form A or $\exists r.D'$, and
4. $I_i^D = I_{i-1}^D$ otherwise.
5. $I^D = I_n^D$, where n is the number of clauses in $d^e(D)$.

Lemma 3. *If I^D is nonempty, then $I^D \models_{\mathbf{D}} C_i^D$ for all clauses C_i^D in $d^e(D)$.*

Proof. We validate that for each C_i^D we have $I^D \models_{\mathbf{D}} C_i^D$. Observe that because of how $d^e(D)$ is defined, every clause in $d^e(D)$ contains either no negative definer literal or $\neg D$ is the only negative definer literal (no clauses with more than one negative definer literal can be derived). This means, for any two clauses $C_i^D, C_j^D \in d^e(D)$, the side conditions of the rules are satisfied (the union never has more than one negative definer literal). We do the proof by contradiction. Assume i is the smallest i with $I^D \not\models_{\mathbf{D}} C_i^D$.

1. If the maximal literal in C_i^D is of the form A or $\exists r.D'$, then the clause is satisfied due to Step 3 in the construction of I^D, which contradicts our assumption.
2. If the maximal literal in C_i^D is of the form $\neg A$, we have $I^D \not\models \neg A$ and therefore $I^D \models A$. This means there must be a clause C_j^D where A is maximal in C_j^D and $I_j^D \not\models C_j^D \setminus \{A\}$, otherwise A is not added to I^D. But then, due to the resolution rule, we also have a clause $C = (C_i^D \cup C_j^D) \setminus \{A, \neg A\}$, which is also in $d^e(D)$. Since $\prec_C$ is the multiset extension of the ordering between literals, C is smaller than C_i^D, since $\neg A \in C_i^D$ and $\neg A$ is larger than all elements in C ($\neg A$ is maximal in C_i^D).

   Since both $I^D \not\models_{\mathbf{D}} C_j^D \setminus \{A\}$ and $I^D \not\models_{\mathbf{D}} C_i^D \setminus \{\neg A\}$, we have $I^D \not\models_{\mathbf{D}} C$. Because C belongs to $d^e(D)$ and is smaller than C_i^D, there is a $k < i$ with $C = C_k^D$, which contradicts our initial assumption that i is the smallest i with $I^D \not\models_{\mathbf{D}} C_i^D$.
3. If the maximal literal in C_i^D is of the form $\forall r.D'$, we have $I^D \not\models_{\mathbf{D}} C_i^D \setminus \{\forall r.D'\}$ and $I^D \not\models_{\mathbf{D}} \forall r.D'$. The only way the latter can be true is due to a literal $\exists r.D_2 \in I^D$ with $D_2 \not\sqsubseteq_D D'$. If D_2 is not subsumed by D', $\exists r.D_2$ is a counter-example for $\forall r.D'$.

   If $\exists r.D_2 \in I^D$, there must be a clause C_j^D such that the maximal literal in C_j^D is $\exists r.D_2$ and $I_j^D \not\models_{\mathbf{D}} C_j^D$. Because of the role propagation rule, we then also have a clause $C_k^D = (C_i^D \cup C_j^D \cup \{\exists r.D_3\}) \setminus \{\forall r.D', \exists r.D_2\}$, where D_3 represents $D' \sqcap D_2$. In our ordering, $\exists r.D_3$ is smaller than both $\forall r.D'$ and $\exists r.D_2$, and therefore $C_k^D \prec_C C_j^D$, $C_k^D \prec_C C_i^D$ and $k < j < i$. We obtain that $I^D \not\models_{\mathbf{D}} C_k^D$ because (i) $I^D \not\models_{\mathbf{D}} C_i^D \setminus \{\forall r.D'\}$, (ii) $I^D \not\models_{\mathbf{D}} C_j^D \setminus \{\exists r.D_2\}$, and (iii) $I^D \not\models_{\mathbf{D}} \exists r.D_3$ (for else $I_j^D \models_{\mathbf{D}} \exists r.D_2$ as $D_3 \sqsubseteq_D D_2$ and $C_k^D \prec_C C_j^D$, and $\exists r.D_3$ cannot be maximal in any clause larger than C_j^D). However, $I^D \not\models_{\mathbf{D}} C_k^D$ contradicts our initial assumption. □

Based on I^D, we construct the candidate model $\mathcal{I}_c = \langle \Delta^{\mathcal{I}_c}, \cdot^{\mathcal{I}_c} \rangle$ with:

- $\Delta^{\mathcal{I}_c} = \{x_D \mid D \in \mathbf{D}$ and I^D is not empty$\}$,
- for every atomic concept A, $A^{\mathcal{I}_c} = \{x_D \mid A \in I^D\}$, and
- for every role r, $r^{\mathcal{I}_c} = \{(x_{D_1}, x_{D_2}) \mid \exists r.D_2 \in I^{D_1}$ *and* I^{D_2} *is nonempty*$\}$.

Lemma 4. *If* $\bot \notin N_s$*, then* $\mathcal{I}_c$ *is a model of* N_s*.*

Proof. We already established that I^D contains all literal concepts that have to hold in order to satisfy the set of clauses $d^e(D)$. All other clauses in N_s are satisfied by I^D as well, since they are all of the form $\neg D' \sqcup C$, and either $D' \notin I^D$, or $\neg D \sqcup D' \in d^e(D)$, and resolution on D' results in $\neg D \sqcup C \in d^e(D)$.

Observe that I^D only contains literals of the form A or $\exists r.D$, which means we do not have to check satisfaction of literals of the form $\forall r.D$. The candidate model is constructed in such a way that for each nonempty I^D, we have a domain element x_D that satisfies all atomic concepts A in I^D. If I^D is empty, it means that we have a unit clause $\neg D$ which is equivalent to $D \sqsubseteq \bot$, and $D^{\mathcal{I}_c}$ should be empty, which is also ensured by the model construction. Therefore we only have to show that, if I^D is nonempty, every existential role restriction $\exists r.D' \in I^D$ holds in $\mathcal{I}_c$ for x_D as well.

- If $I^{D'}$ is not empty, there is a domain element $x_{D'}$ and $(x_D, x_{D'}) \in r^{\mathcal{I}_c}$. Therefore $\exists r.D'$ is satisfied for x_D.
- If $I^{D'}$ is empty, there is no domain element $x_{D'}$, and $\exists r.D'$ is not satisfied by x_D. This can only be the case if $\neg D' \in N_s$. Since we assume $\exists r.D' \in I^D$, there is a clause C_i^D, where $\exists r.D'$ is the maximal literal and $I_i^D \not\models_{\mathbf{D}} C_i^D$. But then, due to the existential role elimination rule and because $\neg D' \in N_s$, there is also the smaller clause $C = C_i^D \setminus \{\exists r.D'\}$. But if $I^D \models C$, then $\exists r.D'$ is not in I^D, which contradicts our assumption that $\exists r.D' \in I^D$. □

We can now prove refutational soundness and completeness of the decision procedure, i.e. Theorem 2.

Proof. Soundness was already established in Lemma 1. Therefore if $N \vdash \bot$, N is unsatisfiable. Suppose $N \not\vdash \bot$. Then we can construct a model for N_s (N saturated by $RES_{\mathcal{ALC}}$) using the method described above (Lemma 4), and since N_s is equi-satisfiable with N (Lemma 1), N is satisfiable. □

7 Correctness of the Uniform Interpolation Method

In order to prove the correctness of our uniform interpolation method, we have to show that every consequence $C \sqsubseteq D$ in the desired signature is preserved by the uniform interpolant. For Phase 2, we use our decision procedure to show that these consequences are preserved by $ELIM_{Res}(N, A)$. This is done by generating a set of clauses M for each consequence $C \sqsubseteq D$, such that $N \models C \sqsubseteq D$ iff $N \cup M$ is unsatisfiable. We first show that for any clause set M over the desired signature, $N \cup M$ is satisfiable iff $ELIM_{Res}(N, A) \cup M$ is satisfiable.

Lemma 5. *For any concept symbol A and any sets of clauses N and M, such that $A \notin sig(M)$, let N_s be the result of saturating $N \cup M$ and N_s' be the result of saturating $ELIM_{Res}(N, A) \cup M$ using $RES_{\mathcal{ALC}}$. It is possible to create a candidate model for N_s' iff it is possible to create a candidate model for N_s.*

Proof. We define the orderings $\prec_L$ and $\prec_C$ as in the last section with the additional constraint that $\neg B \prec_L A$ for any concept symbol $B \neq A$, A being the concept symbol eliminated in N_s'.

We first point out the following properties of clause sets N saturated using $\mathit{RES}_{\mathcal{ALC}}$ regarding definer symbols D: (i) If $|\mathit{conj}(D)| > 1$ (D is an introduced definer), we have $\neg D \sqcup D_i \in N$ for every $D_i \in \mathit{conj}(D)$ (due to role propagation and possibly subsequent resolution steps). Due to further resolution applications, this implies (ii) for every $D_i \sqsubseteq_D D$, we have $d^e(D) \subseteq d^e(D_i)^{D_i \mapsto D}$, where $d^e(D_i)^{D_i \mapsto D}$ denotes the result of replacing every D_i in $d^e(D_i)$ with D. (iii) There is maximally one definer in $\mathit{conj}(D)$ that occurs under an existential role restriction, and every pair of definers in $\mathit{conj}(D)$ occurs under contexts that allow for their combination via the role propagation rule (role propagation is only applied if at least one literal is a universal role restriction and if the side conditions are not violated). (iv) Every nonempty subset of $\mathit{conj}(D)$ is represented by a definer (this is a consequence of (iii)).

Now, observe that in the proof for Lemma 3 only rule applications on maximal literals and definer literals are needed. Resolution on definer literals is assumed indirectly in the proof by how we define satisfaction for literal sets I^D taking into account $\sqsubseteq_D$. This means it is sufficient to perform inferences on maximal literals or definer literals.

A difference between N_s and N_s' is that N_s' does not contain any clauses using A. If we can show that nevertheless conclusions of resolving on A literals occurring in N_s also occur in N_s', we are done, since in $\mathit{RES}_{\mathcal{ALC}}$ rules are applied unrestricted and in $\mathit{ELIM}_{\mathit{RES}}(N, A)$ only clauses containing A are removed. If A is not crucial in deriving the empty clause, it is safe to remove clauses containing it. If A is crucial, the only derivations we lose when removing clauses containing A are conclusions of inference steps involving A.

For pairs of clauses that do not contain any definers, or that only contain definers that are also in $\mathit{ELIM}_{\mathit{RES}}(N, A)$, their resolvents on A are in N_s' since they are in $\mathit{ELIM}_{\mathit{RES}}(N, A)$. Assume we have a clause $C = \neg D \sqcup C_1 \sqcup C_2 \in N_s$ that is the resolvent of two clauses $\neg D \sqcup C_1 \sqcup A, \neg D \sqcup C_2 \sqcup \neg A \in N_s$, such that $C \not\in N_s'$, and assume further that C is the largest clause according to $\prec_C$ with this property. As already mentioned, D cannot be in $\mathit{ELIM}_{\mathit{RES}}(N, A)$, since otherwise C is also in $\mathit{ELIM}_{\mathit{RES}}(N, A)$. Hence, D can only co-occur with A due to resolution on clauses of the form $\neg D \sqcup D_i$, where D_i co-occurs with A. This means there are two clauses $\neg D \sqcup D_1, \neg D \sqcup D_2 \in N_s$, where at least one of D_1 and D_2 co-occurs with A in a clause (observe that $D \sqsubseteq_D D_1$ and $D \sqsubseteq_D D_2$). We have two cases:

1. $\neg D_1 \sqcup C_1 \sqcup A, \neg D_1 \sqcup C_2 \sqcup \neg A \in N_s$. Then $\neg D_1 \sqcup C_1 \sqcup C_2 \in N_s'$ (due to our assumption that C is the largest resolvent not in N_s'), and due to resolution on D_1 we have $C \in N_s'$, which contradicts our assumption that $C \not\in N_s'$.
2. $\neg D_1 \sqcup C_1 \sqcup A, \neg D_2 \sqcup C_2 \sqcup \neg A \in N_s$. Since at least one definer is not in $\mathit{ELIM}_{\mathit{RES}}(N, A)$ (otherwise D would be in $\mathit{ELIM}_{\mathit{RES}}(N, A)$ as well), there must be clauses $\neg D_1' \sqcup C_1 \sqcup A$ and $\neg D_1 \sqcup D_1'$. But then, due to (iv), we also have a definer D' representing $D_1' \sqcap D_2$, and $D \sqsubseteq_D D'$. Due to our assumption, $\neg D' \sqcup C_1 \sqcup C_2 \in N_s'$. Due to (ii), we also have $\neg D \sqcup C_1 \sqcup C_2 = C \in N_s'$, thus contradicting our assumption that $C \not\in N_s'$. □

Now we can prove Theorem 3, which states that for any concept symbol A and clause set N, $\mathit{ELIM}_{\mathit{Res}}(N, A)$ can be computed in finitely bounded time and preserves all consequences over $\mathit{sig}(N) \setminus \{A\}$.

Proof. The fact that $\mathit{ELIM}_{\mathit{Res}}(N, A)$ can always be computed in finitely bounded time follows from Lemma 2. Since in $\mathit{ELIM}_{\mathit{Res}}(N, A)$ all clauses containing A are removed, all symbols in $\mathit{ELIM}_{\mathit{Res}}(N, A)$ are either definers or in $\mathit{sig}(N) \setminus \{A\}$. Hence we only have to check the second condition of the definition of uniform interpolants: $\mathit{ELIM}_{\mathit{Res}}(N, A) \models C \sqsubseteq D$ iff $N \models C \sqsubseteq D$, for any $\mathcal{ALC}$ concept subsumption not containing A.

$N \models C \sqsubseteq D$ can be proven by showing that $C \sqcap \neg D$ is unsatisfiable in N, or by showing that $N' = N \cup \{\top \sqsubseteq \exists r^*.(C \sqcap \neg D)\} \models \bot$, where r^* is a new role not occurring in N. Set $M = \mathit{clauses}(\{\top \sqsubseteq \exists r^*.(C \sqcap \neg D)\}$. Since $A \notin \mathit{sig}(M)$ and due to Lemma 5, we have $N \cup M \models \bot$ iff $\mathit{ELIM}_{\mathit{Res}}(N, A) \cup M \models \bot$. □

We can now prove the correctness of our method (Theorem 1):

Proof. Because of Theorem 3, Phase 2 of the method computes the clausal representation of the uniform interpolant in finite time, independent of the order in which symbols are processed. The correctness of Phase 3 follows from Theorem 4, which can be proved by easy adaptions of the proofs in [1] and [14]. □

8 Related Work

A different method for computing uniform interpolants involving fixpoint operators is presented in [13]. This method is based on computing most general and most specific concepts for the concept symbols to be eliminated. In order to avoid infinite derivations, the derivation graph of this process is checked for cycles and fixpoint operators are introduced where necessary. This method exploits the properties of normalised $\mathcal{EL}$-TBoxes and is therefore not immediately applicable for $\mathcal{ALC}$-TBoxes.

A different approach for the description logic $\mathcal{ALC}$ was published in [18]. In this approach a tableaux calculus is used to incrementally add logical consequences from the input ontology. The uniform interpolant is approximated by replacing symbols outside of the signature by $\top$. Adding more consequences before this replacement approximates the result better, which leads to an incremental approximation of the uniform interpolant. If two succeeding approximations are logically equivalent, the uniform interpolant has been computed. This requires periodically checking for TBox equivalence, which can be expensive if the input ontology is large. Using resolution provides a way to make the computation more goal-oriented, and our definer-based representation of nested formulae facilitates the detection of cyclic structures.

Our approach was influenced by a method for uniform interpolation for modal logic K presented in [10]. Formulae in modal logic K are syntactic variants of $\mathcal{ALC}$ concepts, for which uniform interpolants are always finite. As in our approach, their method is based on a resolution calculus, but no structural transformation is used. For this reason, it uses a more complex resolution framework, which is able to perform inferences on nested formulae. The same method can be used to compute uniform interpolants of $\mathcal{ALC}$ concepts, but cannot be extended to TBoxes without affecting termination.

$\mathcal{ALC}\mu$ concepts are syntactic variants of formulae in the modal μ-calculus, which is multi-modal logic K extended with fixpoint operators. Existence of uniform interpolants in the modal μ-calculus was proven in [4], and in [5] a method to compute uniform interpolants is presented. We have not investigated yet whether this calculus can be used in the context of TBox interpolation.

The method we developed is closely related to two methods developed in the context of second-order quantifier elimination [7]. In second-order quantifier elimination, the aim is to eliminate existentially quantified predicate symbols in order to translate second-order formulae into equivalent formulae in first-order logic. In uniform interpolation the aim is to eliminate symbols as well, even though it is not required that the result is logically equivalent to the corresponding formula in second-order logic. The generalised Ackermann's Lemma is used in the second-order quantifier elimination system DLS [14]. Our resolution-procedure follows a similar principle as the second-order quantifier elimination method SCAN [6]. The idea to use the generalised version of Ackermann's Lemma for quantifier-elimination in description logics was first presented in [17]. There has been some work on second-order quantifier elimination for modal logics [8,15], but it has not yet been investigated how these methods relate to uniform interpolation in description logics.

9 Conclusion and Future Work

We presented a method for computing uniform interpolants of $\mathcal{ALC}$-ontologies. Since uniform interpolants are not always finitely representable in $\mathcal{ALC}$, our method uses fixpoint operators and expresses interpolants in the description logic $\mathcal{ALC}\mu$. If no fixpoint operators are introduced by our method, the result is actually the uniform interpolant in $\mathcal{ALC}$. In the other cases, we offer ways to approximate the result in $\mathcal{ALC}$. Our method mainly consists of two parts. The first part is based on a set of rules influenced by classical resolution to eliminate concept symbols incrementally. This step introduces new symbols, which are eliminated in the second part of our method exploiting the generalised version of Ackermann's Lemma.

We have a first implementation of our method, using further optimisations like standard redundancy elimination techniques, which we did not present here for space reasons. First experiments with available ontologies look promising in the sense that uniform interpolants can be computed in the majority of cases.

One optimisation we are currently working on is optimal use of fixpoint operators. It is possible to create examples where there is a finite uniform interpolant in $\mathcal{ALC}$, even though our method returns a result with fixpoint operators. This is for example the case if the fixpoint expression is redundant because of a cyclic relation between other concepts in the ontology. Investigating several techniques to deal with this problem is the topic of ongoing research. In the future it would be useful to have a method that only introduces fixpoints if they are strictly necessary, that is, if there is no equivalent representation of the interpolant in pure $\mathcal{ALC}$.

Apart from that, we are currently investigating how further description logic constructs such as inverse roles or nominals can be incorporated into our method. We believe that our method can serve as a basis for uniform interpolation in more expressive description logics, such as for example $\mathcal{ALCHI}$.

References

1. Ackermann, W.: Untersuchungen über das Eliminationsproblem der mathematischen Logik. Mathematische Annalen 110(1), 390–413 (1935)
2. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Handbook of Automated Reasoning, pp. 19–99. Elsevier, MIT Press (2001)
3. Calvanese, D., Giacomo, G.D., Lenzerini, M.: Reasoning in expressive description logics with fixpoints based on automata on infinite trees. In: Proc. IJCAI 1999, pp. 84–89. Morgan Kaufmann (1999)
4. D'Agonstino, G., Hollenberg, M.: Uniform interpolation, automata and the modal μ-calculus. In: AiML, vol. 1, pp. 73–84. CSLI Pub. (1998)
5. D'Agostino, G., Lenzi, G.: On modal μ-calculus with explicit interpolants. J. Applied Logic 4(3), 256–278 (2006)
6. Gabbay, D., Ohlbach, H.J.: Quantifier elimination in second-order predicate logic. In: Proc. KR 1992, pp. 425–435. Morgan Kaufmann (1992)
7. Gabbay, D.M., Schmidt, R.A., Szałas, A.: Second Order Quantifier Elimination: Foundations, Computational Aspects and Applications. College Publ. (2008)
8. Goranko, V., Hustadt, U., Schmidt, R.A., Vakarelov, D.: SCAN is complete for all Sahlqvist formulae. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS 2003. LNCS, vol. 3051, pp. 149–162. Springer, Heidelberg (2004)
9. Grau, B.C., Motik, B.: Reasoning over ontologies with hidden content: The import-by-query approach. J. Artificial Intelligence Research 45, 197–255 (2012)
10. Herzig, A., Mengin, J.: Uniform interpolation by resolution in modal logic. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 219–231. Springer, Heidelberg (2008)
11. Lutz, C., Piro, R., Wolter, F.: $\mathcal{EL}$-concepts go second-order: Greatest fixpoints and simulation quantifiers. In: Proc. DL 2010, pp. 43–54. CEUR-WS.org (2010)
12. Lutz, C., Wolter, F.: Foundations for uniform interpolation and forgetting in expressive description logics. In: Proc. IJCAI 2011, pp. 989–995. AAAI Press (2011)
13. Nikitina, N.: Forgetting in General EL Terminologies. In: Description Logics. Proc. DL 2011. CEUR-WS.org (2011)
14. Nonnengart, A., Szałas, A.: A fixpoint approach to second-order quantifier elimination with applications to correspondence theory. In: Logic at Work, pp. 307–328. Springer (1999)
15. Schmidt, R.A.: The Ackermann approach for modal logic, correspondence theory and second-order reduction. J. Appl. Logic 10(1), 52–74 (2012)
16. Simancik, F., Kazakov, Y., Horrocks, I.: Consequence-based reasoning beyond Horn ontologies. In: Proc. IJCAI 2011, pp. 1093–1098. AAAI Press (2011)
17. Szałas, A.: Second-order reasoning in description logics. J. Appl. Non-Classical Logics 16(3-4), 517–530 (2006)
18. Wang, Z., Wang, K., Topor, R., Zhang, X.: Tableau-based forgetting in $\mathcal{ALC}$ ontologies. In: Proc. ECAI 2010, pp. 47–52. IOS Press (2010)

Abduction in Logic Programming as Second-Order Quantifier Elimination

Christoph Wernhard

Technische Universität Dresden

Abstract. It is known that skeptical abductive explanations with respect to classical logic can be characterized semantically in a natural way as formulas with second-order quantifiers. Computing explanations is then just elimination of the second-order quantifiers. By using application patterns and generalizations of second-order quantification, like literal projection, the globally weakest sufficient condition and circumscription, we transfer these principles in a unifying framework to abduction with three non-classical semantics of logic programming: stable model, partial stable model and well-founded semantics. New insights are revealed about abduction with the partial stable model semantics.

1 Introduction

An abductive explanation is basically a formula X such that for given formulas F, the "background knowledge base", and G, the "observation", it holds that F and X together entail G and, in addition, X satisfies application specific further properties, for example, that it only contains symbols from a given vocabulary and that it is as weak as possible. For classical logic, the semantics of an abductive explanation in this sense can be characterized by a second-order formula as follows:

$$X \equiv \forall \mathit{SymbolsNotAllowedInTheExplanation}\ (F \rightarrow G). \qquad \text{(i)}$$

An explanation X can then be *computed* by performing *second-order quantifier elimination* on the second-order formula, that is, computing a formula which is equivalent to the given second-order formula but does not involve second-order quantifiers. If explanations are constrained to be minimal conjunctions of literals, this scheme also applies, but indirectly: the actual explanations are then obtained as the prime implicants of X. Variants of this understanding of abductive explanations are present in a number of works, e.g., [13, p312ff.],[22,7], but the relationship to second-order quantifier elimination seems to have been made explicit first in [5]. Abduction plays several important roles in logic programming, an area where it has been investigated extensively between the late 80s and the early 2000s [17,3]. Many of these approaches are oriented at deriving methods for computing explanations from methods for evaluating logic programs. Semantic characterizations, e.g., [18,8,23,1], are usually placed aside of methods, related to them by correctness properties and complexity results.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 103–119, 2013.

In contrast, the objective of the present work is to combine the second-order elimination approach with non-monotonic semantics of logic programming, resulting in a characterization of abductive explanations for logic programming semantics that is "constructive" in the sense that it maps the computation of explanations to problems of second-order quantifier elimination. As logic programming semantics we consider the popular stable model semantics and two related three-valued semantics, the well-founded and the partial stable model semantics[1]. We work with representations of these logic programming semantics in classical logic extended by second-order operators, based on known translations [21, Section 3.4.1][16]. Under this view, the stable model semantics appears as circumscription that is applied only to certain *occurrences* of predicates – those that are not subjected to negation as failure. Accordingly, a logic program can be represented by a classical formula where these occurrences are distinguished by special predicate names. A logic programming semantics then corresponds to a logical operator *sem* that is wrapped around a classical representation F of a program, such that $sem(F)$ expands into a formula of classical logic extended by second-order operators. The discrimination between different logic programming semantics is expressed by different such wrapping operators, allowing to embed programs considered under different semantics within a single classical formula. With respect to abduction, only a single entailment relation – classical entailment – is required, in contrast to other generic formalizations such as [8], where the discrimination is done "globally" by specific inference operators.

The link between the inherently classical second-order characterization of abductive explanations displayed above as (i) and the non-classical logic programming semantics will be provided by a lemma that states requirements under which the operators *sem* expressing non-monotonic context are "transparent" for explanations E, that is, it holds that $sem(F) \land E \equiv sem(F \land E)$. In the case of the investigated three-valued semantics, two related versions E, E' of the explanation are required, such that the established relationship is $sem(F) \land E \equiv sem(F \land E')$. To determine explanations with respect to a logic program, the abducibles, that is, the atoms that are allowed in explanations should not be submitted to the closed-world assumption, since, unless they occur in rule heads, they would then be just set to false by the non-monotonic semantics. We take this into account by using generalizations of the considered logic programming semantics that allow to specify a set of ground atoms as *open*, that is, not subjected to the closed-world assumption. These generalizations are quite straightforward: In the underlying representations of these semantics by circumscription, the open atoms just correspond to fixed – in contrast to minimized – predicate instances.

The entailment based notion of abductive explanation sketched at the beginning is called *skeptical* or *cautious*. In contrast, *credulous* or *brave* explanations, are constrained by the requirement that the background knowledge base combined with the explanation is *consistent* with observation. For the well-founded semantics every normal logic program has exactly a single model and thus both

[1] In the sense of [24,16], in contrast to contemporary work by Saccà and Zaniolo where *partial stable model* has been used for a related semantics. See [26, Introduction].

notions coincide. In this paper, we focus on the skeptical view for the other semantics. We consider finite normal ground programs, but the material should generalize to programs with disjunctive heads, negation as failure in the head, and first-order quantification, as indicated in [31,33].

As basic second-order operator we use *literal projection* [29], a generalization of predicate quantification. Its arguments make those symbols explicit that are "not quantified" and it allows, so-to-speak, to quantify just upon predicate occurrences with a specific polarity. The latter feature is used to model the considered three-valued logic programming semantics. The application pattern of second-order quantification in (i) is called *globally weakest sufficient condition (GWSC)* and specified in terms of projection. It is closely related to *weakest sufficient condition* [22,5]. Predicate quantification can be applied to express predicate circumscription [4]. We express circumscription by a dedicated second-order operator with a syntax analogous to projection [33]. We will develop the "constructive" characterizations of abductive explanations for the three considered logic programming semantics in parallel. This framework leads to clear formal conceptualizations of various subtle issues in abduction, such as notions of minimality and handling of negative facts in explanations. For abduction with the partial stable model semantics, the author is not aware of another thorough formal treatment. A distinguishing feature of that semantics is that it can be applied to deliver meaningful explanations for facts being observed as *undefined*.

The rest of this paper is organized as follows: In Sect. 2 the background framework of classical propositional logic extended by certain second-order operators is specified. This is applied in Sect. 3 to characterize the considered logic programming semantics. In Sect. 4 definitions of abductive explanation and related concepts are given and in Sect. 5 the central concept of *globally weakest sufficient condition* is summarized. On this basis, the main results of the paper are developed in Sect. 6: Characterizations of abductive explanations and related concepts with respect to logic programming semantics as formulas with second-order operators. Related works are reviewed in Sect. 7 and possible ways to realize the approach in practice are sketched in the conclusion, Sect. 8. Proofs of the results in the paper and further investigations are provided in [34].

2 Notation and Semantic Framework

Formulas, Literals, Scopes and Predicate Groups. We consider *formulas* of classical propositional logic, extended by operators for projection and circumscription. They are constructed from propositional *atoms*, truth value constants $\top, \bot$, the unary connective $\neg$, binary connectives $\land, \lor, \rightarrow, \leftarrow, \leftrightarrow$, as usual, and the two operators $\mathsf{project}$ and circ to express projection and circumscription. As meta-level notation we use n-ary versions of $\land$ and $\lor$. Based on the premise that the material developed here does in principle generalize to a first-order setting, we speak of propositional atoms, or synonymously Boolean variables, also as 0-ary *predicates*. A *literal* is a pair of an atom and a sign, where we write the positive (negative) literal with atom A as $+A$ ($-A$). The complement of a literal L is denoted by $\overline{L}$. If S is a set of literals, then $\overline{S}$ denotes the set of the complements of the members of S. We call a formula that is an atom or a negated atom a *literal formula*, or, if no ambiguity arises, also briefly a *literal.* A *scope* is a set of literals. We assume a fixed propositional signature whose set of atoms is denoted by ATOMS. The sets of all literals, all positive literals, and all negative literals w.r.t. ATOMS are denoted by ALL, POS, NEG, respectively. An *atom scope* S is a scope such that $S = \overline{S}$. Since a literal is a member of an atom scope if and only if its complement is a member, as a shorthand, we represent an atom scope also just by the set of atoms of its members. To express logic programs and three-valued formulas by classical formulas we use a signature where each "original" predicate is available in different "copies", indicating whether an occurrence is subject to negation as failure or how it contributes to the three-valued reading. These "copies" are gathered into so-called *predicate groups*: In addition to the set of propositional atoms ATOMS, we assume a set of *source atoms* that play the role of atoms in other logics that we will represent in our classical framework. Each source atom A is associated with a number of corresponding atoms $A^0, \ldots, A^n \in \mathsf{ATOMS}$, where the superscripts indicate their *predicate group*. More precisely: We assume that ATOMS can be arranged as $\{A_1^0, A_1^1, \ldots, A_1^n, A_2^0, A_2^1, \ldots, A_2^n, A_3^1, \ldots, A_3^n, \ldots\}$ for some $n \geq 1$. For $k \in \{0, \ldots, n\}$, we call the set of all literals whose atom has superscript k the *predicate group* k, written just as the number k. An atom A_i^k is called the *correspondent* from group k of any atom A_i^j. Analogously we speak of correspondents of literals. An *ungrouped scope* is a scope that contains for each of its members all their correspondents. If no ambiguity arises, we write an ungrouped scope like a scope but with omitting the predicate group superscripts. For example, let $\mathsf{ATOMS} = \{p^0, p^1, q^0, q^1, r^0, r^1\}$. Then $1 = \{{+p^1}, {+q^1}, {+r^1}, {-p^1}, {-q^1}, {-r^1}\}$ is a predicate group, and $1 \cap \mathsf{POS} = \{{+p^1}, {+q^1}, {+r^1}\}$. The correspondent of p^1 from group 0 is p^0. An example for an ungrouped atom scope is $\{{+p^0}, {+q^0}, {+p^1}, {+q^1}, {-p^0}, {-q^0}, {-p^1}, {-q^1}\}$, which can be written as $\{p, q\}$. The atom scope $\{{+p^1}, {+q^1}, {-p^1}, {-q^1}\}$ can be written as $1 \cap \{p, q\}$.

Classical Semantics, Projection and Circumscription. An *interpretation* is a set of literals that contains for all atoms $A \in \mathsf{ATOMS}$ exactly one of $+A$ or $-A$. The satisfaction relation $\models$ between interpretations and formulas is defined with a clause for atoms and for each logical operator. For instance, for all interpretations I, scopes S, atoms A, and formulas F, G it holds that: $I \models A$ *iff* $+A \in I$; $I \models \neg F$ *iff* $I \not\models F$; $I \models F \land G$ *iff* $I \models F$ *and* $I \models G$; $I \models \mathsf{project}_S(F)$ *iff there is an interpretation* J *s.t.* $J \models F$ *and* $J \cap S \subseteq I$; $I \models \mathsf{circ}_S(F)$ *iff* $I \models F$ *and there is no interpretation* J *s.t.* $J \models F$ *and* $J \cap S \subset I \cap S$. Entailment and equivalence are then defined as usual: $F \models G$ *iff for all interpretations* I *it holds that if* $I \models F$ *then* $I \models G$; $F \equiv G$ *iff* $F \models G$ *and* $G \models F$.

The formula $\mathsf{project}_S(F)$ whose semantics has just been defined with the $\models$ relationship is called the *literal projection*, or briefly *projection*, of formula F onto scope S. The *forgetting* in F about S is a notational variant where the scope is considered complementary [29,19]:

$$\mathsf{forget}_S(F) \stackrel{\text{def}}{=} \mathsf{project}_{\mathsf{ALL}-S}(F). \qquad \text{(ii)}$$

Combined with first-order logic, projection generalizes second-order quantification, with respect to propositional logic quantified Boolean formulas (QBFs): A QBF $\exists p\, F$ can be expressed as $\mathsf{forget}_{\{+p,-p\}}(F)$ or as $\mathsf{project}_{\mathsf{ALL}-\{+p,-p\}}(F)$. If S is an *atom* scope, the semantic definition of projection is equivalent to: $I \models \mathsf{project}_S(F)$ *iff there is an interpretation* J *s.t.* $J \models F$ *and* $J \cap S = I \cap S$. *Literal* projection also allows to express, so-to-speak, quantification upon just the positive or negative occurrences of a Boolean variable in a formula. Intuitively, the projection of a formula F onto scope S is a formula that expresses about members of S the same as F, but expresses nothing about other literals. A projection of a propositional formula is equivalent to a formula in negation normal form in which only literals in the projection scope do occur. The latter formula is a *uniform interpolant* of the original formula with respect to the scope. A naive way to construct such a uniform interpolant – or to eliminate the projection operator – is indicated by the following equivalences, where $F[p\backslash\top]$ ($F[p\backslash\bot]$) denotes formula F with all occurrences of atom p replaced by $\top$ ($\bot$): (1.) $\mathsf{forget}_{\{+p,-p\}}(F) \equiv F[p\backslash\top] \lor F[p\backslash\bot]$. (2.) $\mathsf{forget}_{\{+p\}}(F) \equiv F[p\backslash\top] \lor (\neg p \land F)$. (3.) $\mathsf{forget}_{\{-p\}}(F) \equiv (p \land F) \lor F[p\backslash\bot]$. For formulas F and scopes S we define

$$F \Subset S \;\; \textit{iff} \;\; F \equiv \mathsf{project}_S(F). \tag{iii}$$

We use the symbol $\Subset$ also when introducing variables, e.g., "let $F \Subset S$ be a formula" for "let F be a formula such that $F \Subset S$". Projection provides a semantic account for systematically replacing atoms from a given predicate group to their correspondents from another one. Let i, j be different predicate groups. We define

$$\mathsf{rename}_{i\backslash j}(F) \;\stackrel{\text{def}}{=}\; \mathsf{forget}_i(F \land \textstyle\bigwedge_{A^i \in \mathsf{ATOMS}}(A^j \leftrightarrow A^i)). \tag{iv}$$

If F is a propositional formula, then $\mathsf{rename}_{i\backslash j}(F)$ is equivalent to F with all occurrences of atoms from group i replaced by their correspondents from j. We define $\mathsf{rename}_{[i_1\backslash j_1,\, \ldots,\, i_n\backslash j_n]}(F)$ as shorthand for $\mathsf{rename}_{i_n\backslash j_n}(\ldots(\mathsf{rename}_{i_1\backslash j_1}(F))\ldots)$.

The circ operator has the same argument types as $\mathsf{project}$ and has also been semantically defined above. It allows to express variants of parallel predicate circumscription where the effects on each atom are controlled by a scope argument [33]. Atoms that occur just in a *positive* literal in the scope are minimized, atoms that occur just in a *negative* literal are maximized, atoms that occur in *both polarities* are fixed and atoms that do *not at all* occur in the scope are varying. Thus, if F is a formula whose atoms are in disjoint sets P, Q and Z, then the *parallel predicate circumscription of* P *in* F *with fixed* Q *and varied* Z, traditionally written as $\mathrm{CIRC}[F; P; Z]$, can be expressed as $\mathsf{circ}_{(P\cap\mathsf{POS})\cup Q}(F)$.

3 Classically Represented Logic Programming Semantics

We consider finite normal logic programs that are ground, that is, finite sets of rules of the form

$$p \leftarrow q_1,\; \ldots,\; q_m,\; \mathsf{not}\;\; r_1,\; \ldots, \mathsf{not}\;\; r_n, \tag{v}$$

where $m, n \geq 0$ and p, q_i, r_i are source atoms. The *classical representation of a normal logic program* is a classical propositional sentence, obtained from the program by forming the conjunction of its members and replacing each atom by its representative from the indicated group as well as replacing the connectives with classical ones, according to the following schema:

$$p^0 \leftarrow q_1^0 \wedge \ldots \wedge q_m^0 \wedge \neg r_1^1 \wedge \ldots \wedge \neg r_n^1. \tag{vi}$$

Information that was expressed in (v) by the positioning of an atom in a rule head versus the negative body is now captured instead by the predicate group.

Stable Model Semantics. For abductive reasoning we consider generalizations of the established logic programming semantics that allow to specify atoms as *open*, that is, not subjected to the closed world assumption. To this end, the operators that express the logic programming semantics have aside of a classical representation of a logic program also an ungrouped atom scope as argument that specifies the open atoms.[2] The logical operator stable renders the stable model semantics: For ungrouped atom scopes O and formulas F define

$$\mathsf{stable}_O(F) \overset{\text{def}}{=} \mathsf{rename}_{1 \setminus 0}(\mathsf{circ}_{(0 \cap \mathsf{POS}) \cup 1 \cup O}(F)). \tag{vii}$$

The circumscription scope in this definition specifies that all atoms from group 1 as well as all open atoms are fixed, while the remaining atoms from group 0 are minimized. This characterization of stable models in terms of circumscription originates from [21, Section 3.4.1] (see also [20,31]). It is expressed here not as a formula transformation but as a logical operator that expands into a classical formula with projection (for the renaming) and circumscription. The stable operator represents the stable model semantics in the following sense: If F is the classical representation of a normal logic program and O is an ungrouped atom scope, then the stable models of the program w.r.t. O are exactly the sets of atoms obtained by taking the set of the positive literals that are from group 0 in some model of $\mathsf{stable}_O(F)$, followed by dropping their signs and group superscripts. For example, the program $\{p \leftarrow \mathsf{not}\ q\}$ has $\{p\}$ as its single stable model, which can be obtained from the models of $\mathsf{stable}_{\{\}}(p^0 \leftarrow \neg q^1) \equiv (p^0 \wedge \neg q^0)$ as described. With respect to $O = \{q\}$, the program has the two stable models $\{p\}$ and $\{q\}$, corresponding to $\mathsf{stable}_{\{q\}}(p^0 \leftarrow \neg q^1) \equiv (p^0 \wedge \neg q^0) \vee (\neg p^0 \wedge q^0)$.

Partial Stable Model Semantics. Partial stable model and well-founded semantics associate three-valued models with a logic program. Predicate groups can be applied to express the three truth values $\mathsf{F}, \mathsf{U}, \mathsf{T}$ in terms of two truth values: An interpretation I over at least all atoms of groups 0 and 1 is said to *assign* to a source atom p the three-valued truth value F *iff* $I \models (\neg p^0 \wedge \neg p^1)$, U *iff* $I \models (\neg p^0 \wedge p^1)$, and T *iff* $I \models (p^0 \wedge p^1)$. The remaining possibility $I \models (p^0 \wedge \neg p^1)$ does not correspond to a three-valued truth value and models with this combination can be excluded with the axiom

$$\mathsf{cons} \overset{\text{def}}{=} \textstyle\bigwedge_{A^0 \in \mathsf{ATOMS}} (A^1 \leftarrow A^0), \tag{viii}$$

[2] It is well-know that specifying atoms as *open* in this sense can also be encoded in the standard versions of these semantics (see discussion of [15] in Sect. 7).

assuming ATOMS is finite. The logical operator $\mathsf{pstable}$ defined below renders the partial stable model semantics [25] by combining the translation of [16] into programs with stable models semantics with the translation of the stable model semantics shown above. Each of the two translations involves discrimination between two predicate groups, yielding four groups $0, 1, 2, 3$ in combination, which are reduced in the final value of $\mathsf{pstable}$ by renaming to groups 0 and 1. The models of $\mathsf{pstable}_O(F)$ represent the three-valued partial stable models by combining the values of atoms for predicate groups 0 and 1. In the definition of $\mathsf{pstable}$ we write the numbers denoting predicate groups in binary notation to indicate how the two involved translations are combined: The right digit corresponds to the group discrimination required by the translation into stable models, the left digit to the discrimination required by expressing the stable model semantics with circumscription. The arguments of $\mathsf{pstable}$ are like those of stable. For ungrouped atom scopes O and formulas F define

$$\begin{aligned}\mathsf{pstable}_O(F) \overset{\text{def}}{=} \mathsf{rename}_{[10\backslash 00,11\backslash 01]}(\mathsf{circ}_M(\mathsf{cons} \land \mathsf{rename}_{[01\backslash 11]}(F) \land \\ \mathsf{rename}_{[01\backslash 10,00\backslash 01]}(F))),\end{aligned} \tag{ix}$$

where $M = ((00 \cup 01) \cap \mathsf{POS}) \cup 10 \cup 11 \cup O$. To represent values of $\mathsf{pstable}$, we write a conjunction C of literal formulas that contains for each atom $p \in \mathsf{ATOMS} \cap (0 \cup 1)$ either p or $\neg p$ as conjunct and is consistent with cons as pair $\langle \mathcal{T}, \mathcal{F} \rangle$ of two sets of source atoms, analogous to common notation for three-valued interpretations: $\mathcal{T}$ is the set of all p such that p^0 is a conjunct in C, and $\mathcal{F}$ is the set of all p such that $\neg p^1$ is a conjunct in C. For example, $(p^0 \land p^1 \land \neg q^0 \land \neg q^1 \land \neg r^0 \land r^1)$ would be written as $\langle \{p\}, \{q\} \rangle$. Compared to the stable model semantics, the partial stable model semantics yields additional models, caused, e.g., by atoms that are "undefined" since they are exempt from the closed world assumption or since they occur "paradoxically" in the head and negated in the body of some rule.

Example 1 (Partial Stable Model Semantics). Let $F = (p^0 \leftarrow q^0)$ and let $O = \{q\}$. Then (1) $\mathsf{stable}_O(F) \equiv (p^0 \land q^0) \lor (\neg p^0 \land \neg q^0)$ and (2) $\mathsf{pstable}_O(F) \equiv \langle \{\}, \{\} \rangle \lor \langle \{p, q\}, \{\} \rangle \lor \langle \{\}, \{p, q\} \rangle$. The first disjunct in (2), that is, $\langle \{\}, \{\} \rangle$, does not correspond to any disjunct in (1). As an example for a "paradoxical" occurrence consider $F' = (p^0 \leftarrow \neg p^1 \land \neg q^1)$. Then $\mathsf{stable}_{\{\}}(F') \equiv \bot$, that is, F' has no stable model. However, $\mathsf{pstable}_{\{\}}(F') \equiv \langle \{\}, \{q\} \rangle$.

Well-Founded Semantics. An interpretation is called *informationally less-or-equal-than* a second one if and only if each atom assigned to one of the three-valued truth values T or F by the first interpretation is assigned to the same value by the second. Models that are minimal with respect to this relation can be characterized by circumscription upon the scope

$$\mathsf{imin\text{-}scope} \overset{\text{def}}{=} (0 \cap \mathsf{POS}) \cup (1 \cap \mathsf{NEG}). \tag{x}$$

If the models of a formula F satisfy cons, then the *informationally minimal* models of F are the models of $\mathsf{circ}_{\mathsf{imin\text{-}scope}}(F)$. If cons is used together with circumscription upon $\mathsf{imin\text{-}scope}$, it can equivalently be placed inside or outside of the circumscription operator: $\mathsf{circ}_{\mathsf{imin\text{-}scope}}(\mathsf{cons} \land F) \equiv \mathsf{cons} \land \mathsf{circ}_{\mathsf{imin\text{-}scope}}(F)$. Now, well-founded models are exactly the informationally minimal partial stable models [24], allowing to characterize the well-founded semantics as

$$\mathsf{wf}_O(F) \overset{\text{def}}{=} \mathsf{circ}_{\mathsf{imin\text{-}scope}}(\mathsf{pstable}_O(F)). \qquad \text{(xi)}$$

An attractive feature of the well-founded semantics is that each normal logic program has exactly a single model. This property applies also to the generalized variant wf_O with specified open atoms. By the following proposition, the consequences G for which it holds that $G \Subset \mathsf{imin\text{-}scope}$ are for the well-founded semantics exactly the same as for the partial stable model semantics:

Proposition 2 (Consequences under Well-Founded and Partial Stable Model Semantics). *If O is a set of ungrouped atoms, and F, G are formulas such that $G \Subset \mathsf{imin\text{-}scope}$, then $\mathsf{wf}_O(F) \models G$ if and only if $\mathsf{pstable}_O(F) \models G$.*

The precondition $G \Subset \mathsf{imin\text{-}scope}$ of Prop. 2 characterizes exactly those formulas G that involve just truth and falsehood, in contrast to "undefinedness", that is, each three-valued model of G which respects the axiom cons assigns T or F to atoms in the signature, but it never assigns U. This can be proven by considering disjunctive normal forms of formulas $G \Subset \mathsf{imin\text{-}scope}$ and relating their conjunctive clauses to models.

4 Basic Concepts of Abduction

An *abductive setting* gathers the parameters of abductive reasoning problems:

Definition 3 (Abductive Setting). An *abductive setting* is a tuple $\mathfrak{A} = \langle sem, O, S, F, G\rangle$ of (1.) a logical operator *sem* with two arguments (an ungrouped atom scope and a formula), the *programming semantics*, (2.) an ungrouped atom scope O, the *open scope*, (3.) an ungrouped scope $S \subseteq O$, the *explanation scope*, (4.) a formula F, the *background*, and (5.) a formula G, the *observation.*

This is similar to *abductive framework* [17], but here also the observation is included. The *programming semantics* is an operator like stable that specifies the logic programming semantics to be used. The *open scope* specifies the atoms that are to be considered open with respect to the logic programming semantics. The *explanation scope* specifies the vocabulary along with associated polarities that is available for explanations. It is equal to or a subset of the open scope, and thus must not necessarily be an atom scope, that is, it can contain literals but not their complements. *Background* and *observation* are formulas representing the background theory presentation and the observation, respectively.

Since we will focus on explanations that are conjunctions of literals, we provide convenient notation for these: A *conjunctive clause* is a consistent conjunction of literal formulas, with the empty conjunction $\top$ as special case. Let C, D be conjunctive clauses. We write $C \models D$ as $D \subseteq C$, and ($C \models D$ *and* $C \not\equiv D$) as $D \subset C$. A conjunctive clause C is called *positive* (*negative*, resp.) if and only if $C \Subset \mathsf{POS}$ ($C \Subset \mathsf{NEG}$, resp.). In this paper we adhere to the skeptical view of explanations, rendered in the following definition:

Definition 4 (Explanation, Factual Explanation). Let $\mathfrak{A} = \langle sem, O, S, F, G \rangle$ be an abductive setting. An *explanation for* $\mathfrak{A}$ is a formula $H \Subset S \cap 0$ such that $sem_O(F \land H) \models G$. An explanation that is a conjunctive clause is called *factual.*

A positive factual explanation can be combined in a particularly simple way with a logic program: If F is the classical representation of a normal logic program and C is a positive explanation, then $(F \land C)$ is again a classical representation of a normal logic program, the original program with the positive literals of the explanation added as facts. Different ways to combine *negative* literals in explanations with programs are discussed in [34, Sect. D]. Certain abductive settings have the property that conjunctive clauses which extend a factual explanation and are in the explanation scope are also explanations, formally:

Definition 5 (Factual Explanation Monotonicity). An abductive setting $\mathfrak{A} = \langle sem, O, S, F, G \rangle$ is called *factual explanation monotonic* if and only if whenever C is a factual explanation for $\mathfrak{A}$, then any conjunctive clause $D \Subset S \cap 0$ such that $C \subseteq D$ is also a factual explanation for $\mathfrak{A}$.

The following property justifies to represent *all* factual explanations of some abductive setting compactly just by the set of *minimal factual explanations*, that is, those factual explanations that do not properly extend some other explanation:

Definition 6 (Minimal Factual Explanation). Let $\mathfrak{A}$ be an abductive setting. A *minimal factual explanation for* $\mathfrak{A}$ is a factual explanation C for $\mathfrak{A}$ such that there does not exist another factual explanation D for $\mathfrak{A}$ with $D \subset C$.

A further notion of "minimality" for factual explanations is obtained by considering just *complete* explanations, explanations that contain for each atom A^0 occurring in of the explanation scope either A^0 or $\neg A^0$, and compare them with respect to their *positive* member literals: $C \leq D$ *iff* $\mathsf{project}_{\mathsf{POS}}(D) \models \mathsf{project}_{\mathsf{POS}}(C)$. We call factual explanations that are minimal in this sense *smallest.* There is a one-one correspondence of the smallest explanations to a certain subset of the minimal explanations [34, Prop. C27]. Smallest explanations can be combined with the background by adding their positive literals as facts, which yields a normal logic program, and removing from the open scope all members whose atom occurs in the explanation scope, independently of the particular explanation [34, Sect. C,D].

In the literature, it is often required that the combination of explanation and background is consistent. For reasons explicated in [34, Sect. B] this is specified here as a separate property:

Definition 7 (Background Consistent Explanation). An explanation H for an abductive setting with semantics *sem*, open scope O and background F is called *background consistent* if and only if $sem_O(F \land H)$ is satisfiable.

Integrity constraints, that is, rules with empty head, are in the literature on abduction in logic programming often assigned a special role. We consider here just normal logic programs, which, however, allow to encode constraints with respect to the *consistency view* [17] by rules with a head atom that indicates failure and is added negated to the observation [8, Sect. 3].

5 The Globally Weakest Sufficient Condition

The *globally weakest sufficient condition (GWSC)* [33] is the application pattern of second-order quantification by which explanations with respect to classical logic are characterized as in (i) in the introduction. We specify it formally in terms of literal projection, such that also polarity can be constrained:

Definition 8 (Globally Weakest Sufficient Condition). The *globally weakest sufficient condition (GWSC)* of formula G on scope S within formula F, in symbols $\mathsf{gwsc}_S(F, G)$, is defined as $\mathsf{gwsc}_S(F, G) \overset{\text{def}}{=} \neg\mathsf{project}_{\overline{S}}(F \land \neg G)$.

The following alternate characterization provides intuition on the relationship to abductive explanations: The GWSC of G on S within F is the weakest formula $H \Subset S$ such that $F \land H \models G$. More precisely:

Proposition 9 (Alternate Characterization of the GWSC). *For all formulas F, G, H and scopes S it holds that $H \equiv \mathsf{gwsc}_S(F, G)$ if and only if: (1.) $H \Subset S$, (2.) $H \models G$, and (3.) for all formulas $H' \Subset S$ such that $F \land H' \models G$ it holds that $H' \models H$.*

The following property implies that a GWSC on scope S can be expressed as a propositional formula in negation normal form that only involves literals from S:

Proposition 10 (Scope Closedness of the GWSC). *For all formulas F, G and scopes S it holds that $\mathsf{gwsc}_S(F, G) \Subset S$.*

The *GWSC* is closely related to *weakest sufficient conditions (WSCs)*, devised in [22] for propositional logic and adapted to first-order logic in [5]. Aside of the consideration of polarity, GWSCs differ from WSCs in the sense of [22] in that for a given formula and scope only GWSCs are unique up to equivalence [33].

6 Abduction with Logic Programming Semantics

The GWSC basically relates to classical semantics. How can it be applied with non-classical logic programming semantics? Lemma 11 below, about "extension transparency", provides the required link. It states requirements that allow a formula to be moved between the context of the non-classical semantics in the argument of the *sem* operator – where the formula "extends" a logic program – and a classical context outside the *sem* operator. Based on this lemma, we then develop characterizations of abductive explanations in terms of the GWSC for the considered logic programming semantics.

The involved lemma, theorem and propositions will be expressed in a generic way, where the differences relating to the particular semantics are factorized out into three auxiliary concepts that expand differently, depending on the semantics indicated by their first argument. The first of these concepts, CF, represents the circumscribed formulas in the definitions of stable and $\mathsf{pstable}$. It is thus defined for formulas F as $\mathsf{CF}(\mathsf{stable}, F) \overset{\text{def}}{=} F$ and $\mathsf{CF}(\mathsf{pstable}, F) \overset{\text{def}}{=} (\mathsf{cons} \land \mathsf{rename}_{[01\backslash 11]}(F) \land$

$\mathsf{rename}_{[01\backslash 11,00\backslash 01]}(F))$. The second concept, IG, is used to project intermediate results onto specific predicate groups and is defined as $\mathsf{IG}(\mathsf{stable}) \overset{\text{def}}{=} 0$ and $\mathsf{IG}(\mathsf{pstable}) \overset{\text{def}}{=} \mathsf{imin\text{-}scope} = (0 \cap \mathsf{POS}) \cup (1 \cap \mathsf{NEG})$. The third concept, IC, is required for three-valued semantics to express a polarity dependent mapping between the predicate groups in conjunctive clauses of explanations and of intermediate results. For stable the value of IC is the unaltered argument, for $\mathsf{pstable}$ it is obtained by switching the group of all negative literals to 1: IC is defined for conjunctive clauses $C = (\bigwedge_{i=1}^{m} A_i^0 \land \bigwedge_{i=1}^{n} \neg B_i^0)$, where $m, n \geq 0$ and $C \Subset 0$, as $\mathsf{IC}(\mathsf{stable}, C) \overset{\text{def}}{=} C$ and $\mathsf{IC}(\mathsf{pstable}, C) \overset{\text{def}}{=} (\bigwedge_{i=1}^{m} A_i^0 \land \bigwedge_{i=1}^{n} \neg B_i^1)$.

Lemma 11 (Extension Transparency). *Let $sem \in \{\mathsf{stable}, \mathsf{pstable}\}$, let F be a formula, let O be an atom scope, and let G be a formula such that $G \Subset (0 \cap (O \cup \mathsf{NEG})) \cup 1$. Then $sem_O(F \land G) \equiv sem_O(F) \land \mathsf{CF}(sem, G)$.*

We apply this lemma mostly to formulas G satisfying the stronger condition $G \Subset 0 \cap O$, which means that G can be expressed in terms of open atoms from group 0. The weaker precondition in the lemma results in the course of the proof [34]. It will be used in Sect. 8 to justify a way in which stable model computation invoked on the background combined with the negated observation can be applied to compute explanations. Based on Lemma 11, Theorem 12 below can be proven. It shows for the stable model and the partial stable model semantics that factual explanations are – modulo conversion by IC – exactly the conjunctive clauses in the explanation scope that imply the GWSC of the program representation wrapped in the semantics operator and of the observation. For the well-founded semantics, the equivalence to the partial stable model semantics with respect to explanations for "defined" observations is stated, which follows from Prop. 2.

Theorem 12 (Factual Explanation in Terms of GWSC). *Let $\mathfrak{A} = \langle sem, O, S, F, G\rangle$ be an abductive setting. Let $C \Subset 0$ be a conjunctive clause. If $sem \in \{\mathsf{stable}, \mathsf{pstable}\}$, then the following two statements are equivalent:*
1. C is a factual explanation for $\mathfrak{A}$.
2. $C \Subset S$ and $\mathsf{IC}(sem, C) \models \mathsf{gwsc}_{S \cap \mathsf{IG}(sem)}(sem_O(F), G)$.
If $sem = \mathsf{wf}$ and $G \Subset \mathsf{imin\text{-}scope}$, then (1.) is equivalent to:
3. C is a factual explanation for $\langle \mathsf{pstable}, O, S, F, G\rangle$.

Since $\mathsf{gwsc}_S(\mathsf{pstable}(F), G) \equiv \mathsf{gwsc}_S(\mathsf{pstable}(F), \mathsf{cons} \land G)$, in abductive settings with $\mathsf{pstable}$ the observation G can be equivalently replaced by any formula G' such that $(\mathsf{cons} \land G') \equiv (\mathsf{cons} \land G)$. In particular, an observation $(p^0 \land p^1)$, which expresses that p is T, can be replaced by just p^0, and $(\neg p^0 \land \neg p^1)$, which expresses that p is F, by just $\neg p^1$. The following example illustrates a case where the factual explanations with stable differ from those with $\mathsf{pstable}$ and wf.

Example 13 (Abduction with Different Semantics I). Let $\mathfrak{A} = \langle sem, O, S, F, G\rangle$ be an abductive setting, where $O = S = \{a, b\}$, $F = (p^0 \leftarrow a^0 \land b^0) \land (p^0 \leftarrow a^0 \land \neg b^1)$, and $G = p^0$. If $sem = \mathsf{stable}$, there is a single minimal factual explanation for $\mathfrak{A}$, namely a^0. If $sem \in \{\mathsf{pstable}, \mathsf{wf}\}$, there are two: First, $(a^0 \land b^0)$, second $(a^0 \land \neg b^0)$. To see that a^0 is then not an explanation, consider that $\mathsf{pstable}_{\{a,b\}}(F \land a^0) \equiv \langle\{a\}, \{\}\rangle \lor \langle\{a, b, p\}, \{\}\rangle \lor \langle\{a, p\}, \{b\}\rangle$.

The following comprehensive example demonstrates further differences of the three considered logic programming semantics with respect to abduction, in particular a case where a meaningful explanation for a fact being observed as *undefined* is only obtained with the partial stable model semantics.

Example 14 (Abduction with Different Semantics II). Assume a domain with two persons a, b, one of them, b, being "the barber". For $x, y \in \{a, b\}$ let sxy stand for "x shaves y", let mx, fx stand for "x is male" and "x is female", respectively. In addition let ss stand for "barbers are self-shavers". The following program F expresses: "a person that is male and does not shave himself is shaved by b", "if barbers are self-shavers, then b shaves himself", and "all persons are either female or male": $F = (sba^0 \leftarrow ma^0 \land \neg saa^1) \land (sbb^0 \leftarrow mb^0 \land \neg sbb^1) \land (sbb^0 \leftarrow ss^0) \land (fa^0 \leftarrow \neg ma^1) \land (ma^0 \leftarrow \neg fa^1) \land (fb^0 \leftarrow \neg mb^1) \land (mb^0 \leftarrow \neg fb^1)$. Let $\mathfrak{A} = \langle sem, O, S, F, G \rangle$ be an abductive setting, where $O = S = \{ma, mb, ss\}$. Let us first consider the partial stable model semantics, i.e., assume $sem = \mathsf{pstable}$. A distinguishing feature of this semantics is that it allows to compute explanations for the *undefinedness* of observations: Let $G = (\neg sbb^0 \land sbb^1)$. Then $G \not\Subset \mathsf{imin\text{-}scope}$ and G expresses that "sbb is U". As the single minimal factual explanation for $\mathfrak{A}$ we then obtain $(mb^0 \land \neg ss^0)$. Since the well-founded model is a partial stable model, this is also an explanation w.r.t. the well-founded semantics. However, there are explanations w.r.t. the well-founded semantics that are not explanations w.r.t. the partial stable model semantics. Here for example the "empty" explanation $\top$, since the well-founded model of F is $\langle \{\}, \{saa, ss\} \rangle$ in which the value of sbb is U. Notice that in the example only the explanation w.r.t. the partial stable model semantics provides the desired information about the reasons for sbb being undefined, i.e., that the barber is male and that "barbers are self-shavers" is false. For "defined" observations G, i.e., if $G \Subset \mathsf{imin\text{-}scope}$, explanations w.r.t. the partial stable model semantics and the well-founded semantics coincide. In the case $G = sbb^0$, expressing that the value of sbb is T, we obtain ss^0 as single minimal factual explanation. In the case $G = \neg sbb^1$, expressing that the value of sbb is F, we obtain $(\neg mb^0 \land \neg ss^0)$. Let us now consider the stable model semantics, i.e., assume $sem = \mathsf{stable}$. For the observation $G = sbb^0$, the only background consistent minimal factual explanation then is ss^0, coinciding with the partial stable model semantics. However, for the observation $G = \neg sbb^0$, the only minimal factual explanation is just $\neg ss^0$. The dependency of $\neg mb^0$ in the explanation obtained for the partial stable model semantics, introduced through the "paradoxical" rule $(sbb^0 \leftarrow mb^0 \land \neg sbb^1)$, is not taken into account by the stable model semantics.

All the three considered logic programming semantics are factual explanation monotonic, which follows from Theorem 12:

Proposition 15 (Factual Explanation Monotonicity of Considered Logic Programming Semantics). *An abductive setting* $\mathfrak{A} = \langle sem, O, S, F, G \rangle$ *where* $sem \in \{\mathsf{stable}, \mathsf{pstable}, \mathsf{wf}\}$ *is factual explanation monotonic.*

Theorem 12 gives a characterization of factual explanations in terms of conjunctive clausal implicants of some particular GWSC. A straightforward consequence is that *minimal* factual explanations correspond to *prime* implicants of that GWSC, as stated in the following proposition. Recall that a *prime implicant* of a formula F is a conjunctive clause C such that $C \models F$ and there does not exist another conjunctive clause D such that $D \models F$ and $D \subset C$.

Proposition 16 (Minimal Factual Explanations and Prime Implicants). *Let* $\mathfrak{A} = \langle sem, O, S, F, G \rangle$ *be an abductive setting. Let* $C \Subset 0$ *be a conjunctive clause. Then the following two statements are equivalent for* $sem \in \{\mathsf{stable}, \mathsf{pstable}\}$*:*
1. C *is a minimal factual explanation for* $\mathfrak{A}$*.*
2. $\mathsf{IC}(sem, C)$ *is a prime implicant of* $\mathsf{gwsc}_{S \cap \mathsf{IG}(sem)}(sem_O(F), G)$*.*
If $sem = \mathsf{wf}$ *and* $G \Subset \mathsf{imin\text{-}scope}$*, then (1.) is equivalent to:*
3. C *is a minimal factual explanation for* $\langle \mathsf{pstable}, O, S, F, G \rangle$*.*

From Prop. 10 it follows that $\mathsf{gwsc}_{S \cap \mathsf{IG}(sem)}(sem_O(F), G) \Subset S \cap \mathsf{IG}(sem)$, and thus, if $sem = \mathsf{pstable}$, then $\mathsf{gwsc}_{S \cap \mathsf{IG}(sem)}(sem_O(F), G)$ is equivalent to a formula in DNF with only consistent disjuncts, where the positive literal formulas are from group 0 and the negative ones from group 1. To convert such a DNF into prime implicants form, i.e., the disjunction of all its prime implicants, it suffices to remove subsumed conjunctive clauses. The following example illustrates the relationship of prime implicants and minimal explanations for the partial stable model semantics.

Example 17 (Prime Implicants Form with Partial Stable Models). Consider the setting of Examp. 13. Then $\mathsf{gwsc}_{S \cap \mathsf{IG}(\mathsf{pstable})}(\mathsf{pstable}_O(F), G) \equiv (a^0 \land b^0) \lor (a^0 \land \neg a^1) \lor (a^0 \land \neg b^1) \lor (b^0 \land \neg b^1)$, where the latter formula is in prime implicants form. To obtain the minimal factual explanations, we remove the two disjuncts $(a^0 \land \neg a^1)$ and $(b^0 \land \neg b^1)$, which would become inconsistent after renaming from group 1 to group 0. This requirement of consistency is implicit in Prop. 16 with the precondition that $C \Subset 0$ is a conjunctive clause.

7 Related Work

As indicated in the introduction in the context of the second-order characterization (i) of classical abductive explanations, similar characterizations have been formulated in a number of works. With respect to non-monotonic semantics, the author is only aware of a second-order characterization for default logic in [28], where a translation of default abduction problems into QBFs is specified such that the models of the resulting QBF correspond to the explanations. The relationship to second-order quantifier *elimination* is not made explicit there. In [7] a QBF characterization of the existence of consistent abductive explanations with respect to classical propositional logic is shown. Only positive explanations, that is, sets of atoms, are permitted. To achieve this, *literal* projection is encoded as Boolean quantification in [7]. Otherwise, the presented schema is essentially (i) conjoined with a condition that ensures background consistency. In [7] also a QBF representation of the stable model semantics is given, but its interplay with abduction is not considered there. In [8] abduction for stable model and well-founded semantics is formalized and complexity results for associated decision problems are given. The role of QBFs there is that hardness results are proven with translations from decision problems for QBFs with certain quantifier prefixes into the abductive decision problems. Negative literals in explanations are not considered in [8].

Several works on computing *credulous* abductive explanations with respect to the stable model semantics are based on the approach of [18]. Similarities to the present work include the consideration of open abducibles and the relationship of minimal explanations to prime implicants. Computation of skeptical explanations can be performed with the credulous approach in a trivial way: Computing all stable models of the background and possible explanations, independently of the observation, and inspecting these afterwards. In [15] it is shown how the computation of credulous explanations with respect to the stable model semantics can be expressed as computation of stable models of programs with integrity rules. The knowledge base is a normal logic program. To encode that abducibles are open, for each abducible p rules $(p \leftarrow \mathsf{not}\, p')$ and $(p' \leftarrow \mathsf{not}\, p)$ are added, where p' is a fresh symbol. Finally, the observation q is added as an integrity constraint $(\bot \leftarrow q)$. There is a one-one correspondence between stable models of the resulting program and explanations of q. As noted in [23], a major drawback of this method is that it involves the actual computation of *all* explanations, not taking into account that the minimal ones provide a succinct representation of them. A variant of [15] is described in [14], where a generalization of the stable model semantics to rules with literals instead of just atoms, as well as disjunctive heads and negation as failure in the head is considered. Computation of explanations is there encoded similarly to [15], except that the openness of abducibles p is expressed by rules $(p \mid \mathsf{not}\ p \leftarrow \top)$. Minimality with respect to the set of abducibles is taken into account [14, Corollary 3.3], but in a way that just suggests to compute first the models and only afterwards extract explanations and compare them with respect to minimality. In [23], the approach of [18] is improved by discerning redundant explanations. Explanations correspond to sets of *literals*. It is shown that the set of all explanations can be represented by the set of minimal explanations, and that minimal explanations to correspond to prime implicants. Again, only credulous explanations are considered.

A characterization of stable models in terms of circumscription is presented in [10] as a transformation $\mathsf{SM}(F)$ on classical formulas F. In contrast to the stable operator, based on [21], the predicate occurrences that are affected by circumscription are identified in [10] by their syntactic position within the formula, such that classically equivalent formulas are not necessarily equivalent with respect to the logic programming semantics. Interestingly, an analog to Lemma 11 is shown in [10, Sect. 5.1]: $\mathsf{SM}(F \land G) \equiv \mathsf{SM}(F) \land G$ *whenever* G *has no strictly positive occurrences* [read: each occurrence is negative, i.e. is in NEG, or is subjected to negation as failure, i.e., is from group 1] *of intensional predicates* [read: predicates that are not open, i.e., are not in O]. Observe that if 0 and 1 are the only predicate groups, then $\mathsf{NEG} \cup 1 \cup O = (0 \cap (O \cup \mathsf{NEG})) \cup 1$, matching exactly the precondition upon G of Lemma 11.

Abduction with respect the well-founded semantics has been elaborated in [1] for programs with a second type of negation, so-called explicit negation, and integrity constraints. A semantic characterization of explanations is specified, and a computation method is described and proven correct. Explicit negation and "coherency" in [1] at least superficially correspond to predicate group 1 and the cons axiom, although a detailed comparison still needs to be done. Concerning abduction with respect to the partial stable model semantics, the present author is not aware of a thorough previous investigation.

8 Conclusion

We have seen that abductive explanations with respect to different logic programming semantics can be characterized semantically as formulas with second-order operators. This provides a solid basis for subtle issues such as abduction with the partial stable model semantics, and, as further described in [34], alternate kinds of minimality, the handling of negative facts in explanations, and abductive consequences. A distinguishing feature of such characterizations is that they can be directly processed by elimination of the second-order operators, that is, computing for a given formula with these operators an equivalent formula without them. Approaches to second-order quantifier elimination include, with respect to first-order and modal logics, the resolution-based SCAN and the direct methods [11]. Of course, with respect to full first-order logic, these methods are inherently incomplete. Further relevant techniques stem from knowledge compilation [30] and SAT solving, where Boolean variable elimination is an important preprocessing technique [6].

From an algorithmic point of view, the elimination approach suggests two possible ways to divide the computation of explanations into subtasks. Consider the computation of all minimal background consistent factual explanations with respect to the stable model semantics. According to Prop. 16, the core expression then is $\mathsf{gwsc}_{S \cap 0}(\mathsf{stable}_O(F), G)$. Explanations can be computed by expanding the gwsc and stable operators, eliminating the resulting second-order quantifiers, and postprocessing the result by computing prime implicants and removing explanations that are not background consistent. A naive implementation that proceeds in this way and allows small experiments is provided with [32][3]. The second way to divide the computation begins with computing $\mathsf{stable}_O(F)$ with a dedicated system for stable models. Lemma 11 justifies to take positive observations into account at this stage: If G contains only positive atoms from group 0, then $\mathsf{gwsc}_{S \cap 0}(\mathsf{stable}_O(F), G) \equiv \neg\mathsf{project}_{S \cap 0}(\mathsf{stable}_O(F) \land \neg G) \equiv \neg\mathsf{project}_{S \cap 0}(\mathsf{stable}_O(F \land \neg G))$. Combinations of stable model computation with second-order quantifier elimination have been developed [9,12], but it needs to be investigated whether they can be used for the computations suggested here.

On the agenda for future work are also further applications of the semantic aspects of the characterizations. For example, relationships to concepts of equivalence of logic programs, in particular abductive equivalence [27] and uniform

[3] Available at `http://cs.christophwernhard.com/toyelim/`

equivalence. Can complexity results be read-off from the characterizations? Are there useful relationships between abduction *with respect to non-monotonic semantics* and the many other applications of GWSC and WSCs [22,5,33] as well as the further similar concept of *perfect rewriting* [2]?

Acknowledgments. The author wishes to thank anonymous reviewers for bringing related work to attention.

References

1. Alferes, J.J., Pereira, L.M., Swift, T.: Abduction in well-founded semantics and generalized stable models via tabled dual programs. Theory and Pract. Log. Program. 4(4), 383–428 (2004)
2. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: View-based query processing: On the relationship between rewriting, answering and losslessness. Theor. Comp. Sci. 371(3), 169–182 (2007)
3. Denecker, M., Kakas, A.C.: Abduction in logic programming. In: Kakas, A.C., Sadri, F. (eds.) Computat. Logic (Kowalski Festschrift). LNCS (LNAI), vol. 2407, pp. 402–436. Springer, Heidelberg (2002)
4. Doherty, P., Łukaszewicz, W., Szałas, A.: Computing circumscription revisited: A reduction algorithm. J. Autom. Reasoning 18(3), 297–338 (1997)
5. Doherty, P., Łukaszewicz, W., Szałas, A.: Computing strongest necessary and weakest sufficient conditions of first-order formulas. In: IJCAI 2001, pp. 145–151. Morgan Kaufmann (2001)
6. 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)
7. Egly, U., Eiter, T., Tompits, H., Woltran, S.: Solving advanced reasoning tasks using quantified Boolean formulas. In: AAAI 2000, pp. 417–422. AAAI Press (2000)
8. Eiter, T., Gottlob, G., Leone, N.: Abduction from logic programs: Semantics and complexity. Theor. Comp. Sci. 189(1-2), 129–177 (1997)
9. Eiter, T., Wang, K.: Semantic forgetting in answer set programming. Artif. Intell. 172(14), 1644–1672 (2008)
10. Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription. Artif. Intell. 175(1), 236–263 (2011)
11. Gabbay, D.M., Schmidt, R.A., Szałas, A.: Second-Order Quantifier Elimination: Foundations, Computational Aspects and Applications. College Publications (2008)
12. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected boolean search problems. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 71–86. Springer, Heidelberg (2009)
13. Inoue, K.: Linear resolution for consequence finding. Artif. Intell. 56(2-3), 301–353 (1992)
14. Inoue, K., Sakama, C.: Negation as failure in the head. J. Log. Program. 35(1), 39–78 (1998)
15. Iwayama, N., Satoh, K.: Computing abduction by using TMS with top-down expectation. J. Log. Program. 44, 179–206 (2000)
16. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding partiality and disjunctions in stable model semantics. ACM Trans. Comput. Log. 7(1), 1–37 (2006)

17. Kakas, A.C., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. In: Gabbay, D.M., Hogger, C.J., Robinson, J.A. (eds.) Handbook of Logic in Artifical Intelligence, vol. 5, pp. 235–324. Oxford University Press (1998)
18. Kakas, A.C., Mancarella, P.: Generalized stable models: A semantics for abduction. In: ECAI 1990, pp. 385–391. Pitman (1990)
19. Lang, J., Liberatore, P., Marquis, P.: Propositional independence – formula-variable independence and forgetting. J. of Artif. Intell. Res. 18, 391–443 (2003)
20. Lifschitz, V.: Twelve definitions of a stable model. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 37–51. Springer, Heidelberg (2008)
21. Lin, F.: A Study of Nonmonotonic Reasoning. Ph.D. thesis, Stanford Univ. (1991)
22. Lin, F.: On strongest necessary and weakest sufficient conditions. Artif. Intell. 128(1-2), 143–159 (2001)
23. Lin, F., You, J.H.: Abduction in logic programming: A new definition and an abductive procedure based on rewriting. Artif. Intell. 140(1/2), 175–205 (2002)
24. Przymusinski, T.: Well-founded semantics coincides with three-valued stable semantics. Fundam. Inform. 13(4), 445–464 (1990)
25. Przymusinski, T.: Stable semantics for disjunctive programs. New Gen. Comput. 9(3/4), 401–424 (1991)
26. Saccá, D., Zaniolo, C.: Deterministic and non-deterministic stable models. J. Log. Comput. 7(5), 555–579 (1997)
27. Sakama, C., Inoue, K.: Equivalence issues in abduction and induction. J. Applied Logic 7(3), 318–328 (2009)
28. Tompits, H.: Expressing default abduction problems as quantified Boolean formulas. AI Commun. 16(2), 89–105 (2003)
29. Wernhard, C.: Literal projection for first-order logic. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 389–402. Springer, Heidelberg (2008)
30. Wernhard, C.: Tableaux for projection computation and knowledge compilation. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS (LNAI), vol. 5607, pp. 325–340. Springer, Heidelberg (2009)
31. Wernhard, C.: Circumscription and projection as primitives of logic programming. In: Tech. Comm. ICLP 2010. LIPIcs, vol. 7, pp. 202–211 (2010)
32. Wernhard, C.: Computing with logic as operator elimination: The ToyElim system. In: WLP 2011, pp. 94–98. Infsys Res. Rep. 1843-11-06, TU Wien (2011)
33. Wernhard, C.: Projection and scope-determined circumscription. J. Symb. Comput. 47(9), 1089–1108 (2012)
34. Wernhard, C.: Abduction in logic programming as second-order quantifier elimination (extended version). Tech. Rep. KRR 13-05, TU Dresden (2013)

Witness Runs for Counter Machines*

Clark Barrett[1], Stéphane Demri[1,2], and Morgan Deters[1]

[1] New York University, USA
[2] LSV, CNRS, France

Abstract. In this paper, we present recent results about the verification of counter machines by using decision procedures for Presburger arithmetic. We recall several known classes of counter machines for which the reachability sets are Presburger-definable as well as temporal logics with arithmetical constraints. We discuss issues related to flat counter machines, path schema enumeration, and the use of SMT solvers.

1 Introduction

Infinite-state systems. Model-checking is a standard approach to verifying properties of computing systems [CGP00] and it is known that dealing with infinity or unboundedness of computational structures leads easily to undecidable verification problems. Such problems include testing boundedness (checking whether a counter in a counter machine takes a finite amount of values) and those dealing with model-checking temporal formulae in which atomic formulae can state properties about unbounded values (e.g., arithmetical constraints about counter values). Roughly speaking, techniques for the verification of infinite-state systems stem from exact methods in which potentially infinite sets of configurations are finitely represented symbolically to semi-algorithms that are designed to behave well in practice. When exact methods can produce decision procedures, this is because an underlying finite structure can be identified in the verification problem. For instance, the set of reachable configurations can be effectively represented symbolically, typically by a formula in Presburger arithmetic, for which satisfiability is known to be decidable [Pre29]. The use of Presburger arithmetic for formal verification has been advocated in [SJ80]. Finiteness can also occur in a more subtle way, as in well-structured transition systems [FS01], for which termination is guaranteed thanks to underlying well-quasi-orderings, see e.g. [Kos82, OW05].

Counter machines. Counter machines are well-known infinite-state systems that have many applications in formal verification. Their ubiquity stems from their use as operational models for several purposes, including for instance for broadcast protocols [FL02], for programs with pointer variables (see [BBH$^+$06]) and for logics for data words, see e.g. [BL10]. However, numerous model-checking

* Work partially supported by the EU Seventh Framework Programme under grant agreement No. PIOF-GA-2011-301166 (DATAVERIF).

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 120–150, 2013.

problems for counter machines, such as reachability, are known to be undecidable. Many subclasses of counter machines admit a decidable reachability problem, such as reversal-bounded counter automata [Iba78] and flat counter automata [CJ98, Boi99, FL02]. These two classes of machines admit reachability sets effectively definable in Presburger arithmetic (assuming some additional conditions, unspecified herein). In general, computing the transitive closures of integer relations is a key step to solve verification problems on counter machines, see e.g. [BW94, CJ98, Fri00, FL02, BIK09].

Flatness. Flat counter machines are counter machines in which each control state belongs to at most one simple cycle (i.e., a cycle without any repetition of edges). Several classes of such flat operational devices have been identified and reachability sets have been shown effectively Presburger-definable for many of them, see e.g. [FO97, CJ98, Boi99, FL02, BIK09]. This provides a decision procedure for the reachability problem, given a prover for Presburger arithmetic validity. Effective semilinearity boils down to check that the effect of a loop can be characterized by a formula in Presburger arithmetic (or in any decidable fragment of first-order arithmetic). The results for flat counter machines can be then obtained by adequately composing formulae for loops and for finite paths. However, this approach, briefly described in this paper, suffers from at least two drastic limitations. First, flatness in counter machines remains a strong restriction on the control graph, though this has been relaxed by considering flattable counter machines, see e.g. [BFLS05, LS05, Ler13] and Section 3.5, where a machine may not itself be flat, but is known to have a flat unfolding with the same reachability set. The second limitation is due to the fact that reachability questions are not the only interesting ones and the verification of properties expressed in dedicated temporal logics is often desirable, see e.g. [DFGvD10].

In this paper, we present a selection of results about the verification of counter machines, at times assuming flatness, from reachability problems to model-checking problems with temporal logics. We follow an approach similar to [Fri00] to translate verification problems into Presburger arithmetic satisfiability. We focus on flattable counter machines and how to compute flat unfoldings by enumerating path schemas while invoking SMT solvers to optimize this enumeration. This part of the paper presents preliminary results, and it will be the subject of a dedicated paper.

Satisfiability Modulo Theories. Deciding Presburger arithmetic fragments is essential to verify properties of programs; see e.g. [Sho79] and [SJ80] for an early use of Presburger arithmetic for formal verification. Most well-known SMT solvers deal with quantifier-free linear integer formulae, also known as quantifier-free linear integer arithmetic ('QF_LIA' in the parlance of SMT-LIB [BST12]). For instance, this includes Z3 [dMB08], CVC4 [BCD$^+$11] and Alt-Ergo [Con12], to cite a few of them. However, dealing with quantifiers is usually a difficult task for SMT solvers that are better tailored to theory reasoning. Many general-purpose SMT solvers (including CVC3, CVC4, Z3, Yices, Alt-Ergo) do accept formulas with quantifiers and they handle them in roughly the same way, through heuristic instantiation. Z3 is unique in that it implements several quantifier-elimination procedures as preprocessing steps, including a procedure for Presburger arithmetic ('LIA' in SMT-LIB). It is worth also mentioning automata-based tools dealing with satisfiability such as MONA [BKR96], LASH [BJW01] or TAPAS [LP09]. Even though Presburger arithmetic admits quantifier elimination, it is known that eliminating quantifiers can be computationally expensive (see e.g., [RL78, Grä88]). Recent developments propose a promising, lazy approach for quantifier elimination [Mon10].

2 Machines with Registers

In this section, we briefly present Presburger arithmetic (PA), the class of Presburger counter systems, and standard subclasses.

2.1 Presburger Arithmetic in a Nutshell

Presburger arithmetic (PA) has been introduced by M. Presburger in [Pre29] where it is shown decidable by quantifier elimination. This decidability result on the theory of addition is regarded today as a key result in theoretical computer science.

Let VAR $= \{\mathsf{x}, \mathsf{y}, \mathsf{z}, \ldots\}$ be a countably infinite set of *variables*. *Terms* are expressions of the form $a_1\mathsf{x}_1 + \cdots + a_n\mathsf{x}_n + k$ where $a_1, \ldots, a_n$ are constant coefficients in $\mathbb{N}$, k is in $\mathbb{N}$ and the x_i's are variables. Variables and terms come with their interpretations when the variables are interpreted by natural numbers. A *valuation* $\mathfrak{v}$ is a map VAR $\rightarrow \mathbb{N}$ and it can be extended to the set of all terms as follows: $\mathfrak{v}(k) = k$, $\mathfrak{v}(a\mathsf{x}) = a \times \mathfrak{v}(\mathsf{x})$ and $\mathfrak{v}(\mathtt{t} + \mathtt{t}') = \mathfrak{v}(\mathtt{t}) + \mathfrak{v}(\mathtt{t}')$ for all terms $\mathtt{t}$ and $\mathtt{t}'$. *Formulae* are defined by the grammar below:

$$\phi ::= \ \mathtt{t} \leq \mathtt{t}' \mid \neg\phi \ \mid \ \phi \wedge \phi \ \mid \ \exists\, \mathsf{x}\, \phi$$

where $\mathtt{t}$ and $\mathtt{t}'$ are terms and $\mathsf{x} \in$ VAR. A formula ϕ is in the *linear fragment* $\overset{\text{def}}{\Leftrightarrow}$ ϕ is a Boolean combination of atomic formulae of the form $\mathtt{t} \leq \mathtt{t}'$. The semantics for formulae in (PA) is defined with the help of the satisfaction relation $\models$ that determines the conditions for the satisfaction of a formula under a given valuation (we omit the Boolean clauses):

- $\mathfrak{v} \models \mathtt{t} \leq \mathtt{t}' \overset{\text{def}}{\Leftrightarrow} \mathfrak{v}(\mathtt{t}) \leq \mathfrak{v}(\mathtt{t}')$,
- $\mathfrak{v} \models \exists\, \mathsf{x}\, \phi \overset{\text{def}}{\Leftrightarrow}$ there is $n \in \mathbb{N}$ such that $\mathfrak{v}[\mathsf{x} \mapsto n] \models \phi$ where $\mathfrak{v}[\mathsf{x} \mapsto n]$ is equal to $\mathfrak{v}$ except that x is mapped to n.

Any formula $\phi(\mathsf{x}_1, \ldots, \mathsf{x}_n)$ whose free variables are among $\mathsf{x}_1, \ldots, \mathsf{x}_n$, with $n \geq 1$, defines a set of n-tuples $[\![\phi(\mathsf{x}_1, \ldots, \mathsf{x}_n)]\!] \overset{\text{def}}{=} \{\langle \mathfrak{v}(\mathsf{x}_1), \ldots, \mathfrak{v}(\mathsf{x}_n)\rangle \in \mathbb{N}^n : \mathfrak{v} \models \phi\}$ which contains all the tuples that make true the formula ϕ by ignoring the irrelevant interpretation of the bound variables and by fixing an arbitrary ordering between the variables. For instance, $[\![\mathsf{x}_1 < \mathsf{x}_2]\!] = \{\langle n, n'\rangle \in \mathbb{N}^2 : n < n'\}$. Let ϕ be a formula $\phi(\mathsf{x}_1, \ldots, \mathsf{x}_n)$ with $n \geq 1$ free variables $\mathsf{x}_1, \ldots, \mathsf{x}_n$. We say that

$[\![\phi]\!]$ is a *Presburger set.* The *satisfiability problem* for (PA) is a decision problem that takes as input a formula ϕ and asks whether there is a valuation $\mathfrak{v}$ such that $\mathfrak{v} \models \phi$. If such a valuation exists, we say that ϕ is *satisfiable.*

Theorem 1 (Presburger Arithmetic Decidability). *[Pre29] The satisfiability problem for (PA) is decidable.*

The satisfiability problem can be solved in triple exponential time [Opp78] by analyzing the quantifier elimination procedure described in [Coo72]. It was shown 2ExpTime-hard in [FR74] and to be in 2ExpSpace in [FR79]. An exact complexity characterization is provided in [Ber80] (double exponential time on alternating Turing machines with linear amounts of alternations).

2.2 Presburger Counter Systems

The systems introduced below are finite-state automata augmented with registers, also known as counters (variables interpreted as natural numbers). Transitions are labelled by arithmetical constraints on counters defined in (PA). A *Presburger counter system* $\mathtt{C} = \langle Q, n, \delta \rangle$ is a structure (see e.g. [DFGvD10, Ler12]) such that

- Q is a nonempty finite set of *control states,*
- $n \geq 1$ is the *dimension* of the system, i.e. the number of counters, we assume that the counters are represented by the variables $\mathsf{x}_1, \ldots, \mathsf{x}_n$,
- δ is the *transition relation* defined as a finite set of triples of the form $\langle q, \phi, q' \rangle$, where q, q' are control states and ϕ is a Presburger formula whose free variables are among $\mathsf{x}_1, \ldots, \mathsf{x}_n, \mathsf{x}'_1, \ldots, \mathsf{x}'_n$. Prime variables are intended to be interpreted as the next values of the unprimed variables.

Figure 1 contains a Presburger counter system $\mathtt{C}$ such that inc(i) [resp. dec(i)] stands for the formula that increments [resp. decrements] the counter x_i and keeps unchanged the other counters. Formulae zero(i) tests if counter x_i is equal to zero but it has no effect on the counters.

Elements $t = \langle q, \phi, q' \rangle \in \delta$ are called *transitions* and are often represented by $q \xrightarrow{\phi} q'$. A *configuration* of the Presburger counter system $\mathtt{C} = \langle Q, n, \delta \rangle$ is a pair $\langle q, \boldsymbol{x} \rangle \in Q \times \mathbb{N}^n$. Given two configurations $\langle q, \boldsymbol{x} \rangle$, $\langle q', \boldsymbol{x'} \rangle$ and a transition $t = q \xrightarrow{\phi} q'$, we write $\langle q, \boldsymbol{x} \rangle \xrightarrow{t} \langle q', \boldsymbol{x'} \rangle$ whenever $\mathfrak{v} \models \phi$ (in (PA)) and for every $i \in [1, n]$, $\mathfrak{v}(\mathsf{x}_i) \stackrel{\text{def}}{=} \boldsymbol{x}(i)$ and $\mathfrak{v}(\mathsf{x}'_i) \stackrel{\text{def}}{=} \boldsymbol{x'}(i)$. Given a Presburger counter system $\mathtt{C}$, its *transition system* $\mathfrak{T}(\mathtt{C}) = \langle \mathfrak{S}, \rightarrow \rangle$ is a graph with $\mathfrak{S} = Q \times \mathbb{N}^n$ and $\rightarrow \subseteq \mathfrak{S} \times \mathfrak{S}$ such that $\langle \langle q, \boldsymbol{x} \rangle, \langle q', \boldsymbol{x'} \rangle \rangle \in \rightarrow \stackrel{\text{def}}{\Leftrightarrow}$ there exists a transition $t \in \delta$ such that $\langle q, \boldsymbol{x} \rangle \xrightarrow{t} \langle q', \boldsymbol{x'} \rangle$. As usual, $\xrightarrow{*}$ denotes the reflexive and transitive closure of the binary relation $\rightarrow$. The binary relation $\xrightarrow{*}$ is also called the *reachability relation* of $\mathtt{C}$ and it is sometimes written $\mathtt{Reach}_{\mathtt{C}}$. Similarly, we write $\mathtt{Reach}_{\mathtt{C}}(\langle q, \boldsymbol{x} \rangle)$ to denote the *reachability set* $\{\langle q', \boldsymbol{x'} \rangle \in \mathfrak{S} : \langle q, \boldsymbol{x} \rangle \xrightarrow{*} \langle q', \boldsymbol{x'} \rangle\}$. A *run* ρ is a nonempty (possibly infinite) sequence $\rho = \langle q_0, \boldsymbol{x_0} \rangle, \ldots, \langle q_k, \boldsymbol{x_k} \rangle, \ldots$ of configurations such that two consecutive configurations are in the relation $\rightarrow$ from $\mathfrak{T}(\mathtt{C})$.

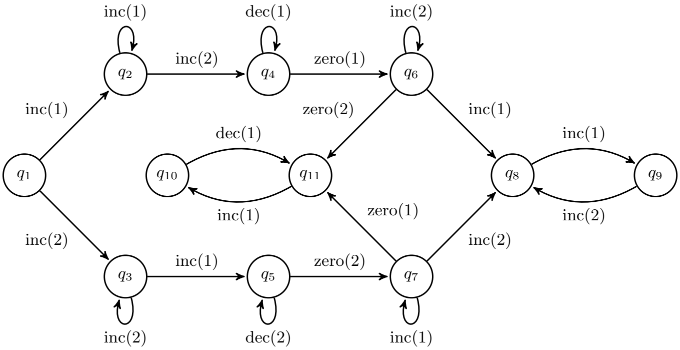


Fig. 1. A Presburger counter system

Most verification problems on Presburger counter systems are known to be undecidable since they include Minsky machines [Min67, Chapter 11] (see also [Min61, Section 3]) that are Turing-complete, even if restricted to two counters [Min67, Chapter 14]. The introduction of *program machines with registers* in [Min67], nowadays best known as Minsky machines, has been motivated by proposing an alternative to Turing machines that is closely related to programs.

2.3 Decision Problems

In this section, we recall several standard decision problems about Presburger counter systems. They are mainly related to reachability questions (problems related to temporal logics are introduced in Section 4).

Reachability problem:

Input: a Presburger counter system $\mathtt{C}$ and configurations $\langle q_0, \boldsymbol{x_0} \rangle$ and $\langle q_f, \boldsymbol{x_f} \rangle$.
Question: is there a finite run from $\langle q_0, \boldsymbol{x_0} \rangle$ to $\langle q_f, \boldsymbol{x_f} \rangle$?

Control state reachability problem:

Input: a Presburger counter system $\mathtt{C}$, a configuration $\langle q_0, \boldsymbol{x_0} \rangle$ and a state q_f.
Question: is there a finite run with initial configuration $\langle q_0, \boldsymbol{x_0} \rangle$ and whose final configuration has control state q_f?

Other verification problems on Presburger counter systems are worth mention, though not discussed herein, including the *control state repeated reachability problem* and the *termination problem.*

2.4 Some Classes of Presburger Counter Systems

In this section, we introduce classes of Presburger counter systems by restricting the general definition provided above. Additional requirements can be of distinct nature: restriction on syntactic resources (number of counters, etc.), restriction on the control graph (e.g., flatness) and semantical restrictions (reversal-boundedness, etc.).

Counter systems. A *counter system* $\mathtt{C} = \langle Q, n, \delta \rangle$ is a Presburger counter system such that for each transition $t = q \xrightarrow{\phi} q' \in \delta$, ϕ can be written as $\phi_g \wedge \phi_u$, where ϕ_g (guard) is a Boolean combination of atomic formulae of the form $\mathtt{t} \leq \mathtt{t}'$ built over $\mathsf{x}_1, \ldots, \mathsf{x}_n$ and ϕ_u (update) is a formula of the form $\bigwedge_{i \in [1,n]} \mathsf{x}'_i = \mathsf{x}_i + \boldsymbol{b}(i)$ where $\boldsymbol{b} \in \mathbb{Z}^n$. The transition t is also written $q \xrightarrow{\langle \phi_g, \boldsymbol{b} \rangle} q'$. Minsky machines [Min67] are counter systems such that each update can change at most one counter and the guards are restricted to $\top$ and to zero-tests. The Presburger counter system in Figure 1 is indeed a counter system with the above meaning.

A *vector addition system with states* [KM69] (VASS for short) is a counter system such that all the transitions are of the form $q \xrightarrow{\langle \top, \boldsymbol{b} \rangle} q'$. So, a VASS can be represented by a tuple $\mathcal{V} = \langle Q, n, \delta \rangle$ where Q is the finite set of control states and δ is a finite subset of $Q \times \mathbb{Z}^n \times Q$. A famous result states that the reachability problem for VASS is decidable [May84, Kos82, Ler09]. It has been the subject of the book [Reu90] and its proof requires many non-trivial steps involving graph theory, logic and theory of well-quasi-orderings. Nevertheless, the exact complexity of the reachability problem is open: we know it is ExpSpace-hard [Lip76] and no primitive recursive upper bound exists. In [Ler09], existence of semilinear separators in case of non-reachability in VASS leads to promising developments.

A Note to the Reader. Counter systems are the main class of Presburger counter systems considered in this document. However, we are aware that the current term might be confusing: when we really want to mean the full class of systems, we will use the more general term ‘Presburger counter system.’

Reversal-bounded counter systems. In this section, we consider counter systems for which the atomic formulae in guards are of the form $\mathtt{t} \leq k$ or $\mathtt{t} \geq k$ with $k \in \mathbb{Z}$ and $\mathtt{t}$ is of the form $\sum_i a_i \mathsf{x}_i$ with the a_i's in $\mathbb{Z}$. There is no real restriction with the class introduced earlier except that we require that atomic formulae occur in a certain way.

A *reversal* for a counter occurs in the run of some counter system when there is an alternation from nonincreasing mode to nondecreasing mode and vice-versa. Below, we propose a slight generalization from [BD11] that captures the notion of reversal-boundedness from [Iba78]; reversal-boundedness applies to counters but also to terms occurring in guards. Let $\mathtt{C} = \langle Q, n, \delta \rangle$ be a counter system and $\mathtt{T}$ be a finite set of terms including $\{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$. From a run $\rho = \langle q_0, \boldsymbol{x_0} \rangle, \langle q_1, \boldsymbol{x_1} \rangle, \ldots$ of $\mathtt{C}$, in order to describe the behavior of counters and terms varying along ρ, we define a sequence of *mode vectors* $\mathfrak{md}_0, \mathfrak{md}_1, \ldots$ (of the same length as ρ) such that each $\mathfrak{md}_i$ has profile $\mathtt{T} \rightarrow \{\nearrow, \searrow\}$. Intuitively, each value in a mode vector records whether a term is currently in an increasing phase or in an decreasing phase (this includes the counters themselves as in standard reversal-boundedness). Given $\mathtt{t} = \sum_i a_i \mathsf{x}_i$ and a counter vector $\boldsymbol{x}$, we write

$\boldsymbol{x}(\mathtt{t})$ to denote the integer $\sum a_i \boldsymbol{x}(i)$. We are now ready to define the sequence $\mathfrak{md}_0, \mathfrak{md}_1, \ldots$ By convention, $\mathfrak{md}_0$ is the constant map $\nearrow$. For every $j \geq 0$ and $\mathtt{t} \in \mathtt{T}$, we have $\mathfrak{md}_{j+1}(\mathtt{t}) \overset{\text{def}}{=} \mathfrak{md}_j(\mathtt{t})$ when $\boldsymbol{x}_j(\mathtt{t}) = \boldsymbol{x}_{j+1}(\mathtt{t})$, $\mathfrak{md}_{j+1}(\mathtt{t}) \overset{\text{def}}{=} \nearrow$ when $\boldsymbol{x}_{j+1}(\mathtt{t}) - \boldsymbol{x}_j(\mathtt{t}) > 0$ and $\mathfrak{md}_{j+1}(\mathtt{t}) \overset{\text{def}}{=} \searrow$ when $\boldsymbol{x}_{j+1}(\mathtt{t}) - \boldsymbol{x}_j(\mathtt{t}) < 0$. Let $Rev_{\mathtt{t}} = \{j \in [0, \mathrm{len}(\rho) - 1] : \mathfrak{md}_j(\mathtt{t}) \neq \mathfrak{md}_{j+1}(\mathtt{t})\}$; we say that ρ is r-$\mathtt{T}$-*reversal-bounded* for some $r \geq 0 \overset{\text{def}}{\Leftrightarrow}$ for all $\mathtt{t} \in \mathtt{T}$, $\mathrm{card}(Rev_{\mathtt{t}}) \leq r$. Given a counter system $\mathtt{C}$, we write $\mathtt{T}_{\mathtt{C}}$ to denote the set of terms $\mathtt{t}$ occurring in atomic formulae of the form $\mathtt{t} \sim k$ with $\sim \in \{\leq, \geq\}$ augmented with the counters in $\{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$. An initialized counter system $\langle \mathtt{C}, \langle q, \boldsymbol{x} \rangle \rangle$ is *reversal-bounded* $\overset{\text{def}}{\Leftrightarrow}$ there is $r \geq 0$ such that every run from $\langle q, \boldsymbol{x} \rangle$ is r-$\mathtt{T}_{\mathtt{C}}$-reversal-bounded. When $\mathtt{T}$ is reduced to $\{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$, $\mathtt{T}$-reversal-boundedness is equivalent to reversal-boundedness from [Iba78]. Note that the counter system in Figure 1 is $\{\mathsf{x}_1, \mathsf{x}_2\}$-reversal-bounded from any initial configuration of the form $\langle q_1, \boldsymbol{x}_{\boldsymbol{0}} \rangle$.

Compared to the subclasses considered so far, reversal-bounded counter systems are augmented with an initial configuration so that existence of the bound r is relative to the initial configuration. Secondly, this class is not defined from the class of counter systems by imposing syntactic restrictions but rather semantically. The main property related to reversal-bounded counter systems is the result below.

Theorem 2. *[Iba78, BD11] Given a counter system* $\mathtt{C}$*,* $r \geq 0$ *and control states* q, q'*, one can effectively compute a Presburger formula* $\phi_{q,q'}(\mathsf{x}_1, \ldots, \mathsf{x}_n, \mathsf{y}_1, \ldots, \mathsf{y}_n)$ *such that for all valuations* $\mathfrak{v}$*, we have* $\mathfrak{v} \models \phi$ *iff there is an* r*-*$\mathtt{T}_{\mathtt{C}}$*-reversal-bounded run from* $\langle q, \langle \mathfrak{v}(\mathsf{x}_1), \ldots, \mathfrak{v}(\mathsf{x}_n) \rangle \rangle$ *to* $\langle q', \langle \mathfrak{v}(\mathsf{y}_1), \ldots, \mathfrak{v}(\mathsf{y}_n) \rangle \rangle$*.*

So, bounding the number of reversals in runs allows to characterize the reachability sets by computing Presburger formulae. This approach can be generalized to richer models, see e.g., [HR87, FS08, HL11].

Affine Presburger counter systems. Now, we present the class of *affine Presburger counter systems* that substantially extends the class of counter systems by allowing any guard that can be defined in (PA) and by giving the possibility to have affine updates. A partial function f from $\mathbb{N}^n$ to $\mathbb{N}^n$ is *affine* $\overset{\text{def}}{\Leftrightarrow}$ there exist a matrix $\mathtt{A} \in \mathbb{Z}^{n \times n}$ and $\boldsymbol{b} \in \mathbb{Z}^n$ such that for every $\boldsymbol{a} \in \mathrm{dom}(f)$, we have $f(\boldsymbol{a}) = \mathtt{A}\boldsymbol{a} + \boldsymbol{b}$. f is *Presburger-definable* $\overset{\text{def}}{\Leftrightarrow}$ the graph of f is a Presburger set (binary relation).

A Presburger counter system $\mathtt{C} = \langle Q, n, \delta \rangle$ is *affine* when for every transition $q \xrightarrow{\phi} q' \in \delta$, $[\![\phi]\!]$ is affine and there is a triple $\langle \phi_g, \mathtt{A}, \boldsymbol{b} \rangle$ such that ϕ_g (guard) is a formula in (PA) with free variables among $\mathsf{x}_1, \ldots, \mathsf{x}_n$ and $[\![\phi]\!] = \{\langle \boldsymbol{x}, \boldsymbol{x'} \rangle \in \mathbb{N}^{2n} : \boldsymbol{x'} = \mathtt{A}\boldsymbol{x} + \boldsymbol{b} \text{ and } \boldsymbol{x} \in [\![\phi_g]\!]\}$. The formula ϕ_g represents the guard of the transition and the pair $\langle \mathtt{A}, \boldsymbol{b} \rangle$ is the deterministic update function. Such a triple $\langle \phi_g, \mathtt{A}, \boldsymbol{b} \rangle$ is called an *affine update* and we also write $[\![\langle \phi_g, \mathtt{A}, \boldsymbol{b} \rangle]\!]$ to denote $[\![\phi]\!]$. Observe that one can decide whether a Presburger formula ϕ satisfies that $[\![\phi]\!]$ is affine [DFGvD10, Proposition 3]. Furthermore, counter systems are affine counter systems in which the only matrix is identity. This class of Presburger counter systems has been introduced in [FL02].

Observe that given $t = q \xrightarrow{\langle \phi_g, \mathtt{A}, \boldsymbol{b} \rangle} q'$, there is a Presburger formula $\varphi(\overline{\mathsf{x}}, \overline{\mathsf{x}'})$ such that for every $\mathfrak{v}$, we have $\mathfrak{v} \models \varphi$ iff $\langle q, \langle \mathfrak{v}(\mathsf{x}_1), \ldots, \mathfrak{v}(\mathsf{x}_n) \rangle \rangle \xrightarrow{t} \langle q', \langle \mathfrak{v}(\mathsf{x}'_1), \ldots, \mathfrak{v}(\mathsf{x}'_n) \rangle \rangle$. Here is the witness formula that encodes the one-step relation:

$$\phi_g(\overline{\mathsf{x}}) \wedge \bigwedge_{i \in [1,n]} (\mathsf{x}'_i = \sum_j \mathtt{A}(i,j) \mathsf{x}_j + \boldsymbol{b}(i))$$

Note that the composition of affine updates is still an affine update.

Presburger counter systems with octagonal constraints. A *Presburger counter systems with octagonal constraints* is such that for each transition $q \xrightarrow{\phi} q' \in \delta$, the formula ϕ is a conjunction of atomic formulae of the form $\pm \mathsf{y} \pm \mathsf{z} \leq k$ where y, z are variables among $\mathsf{x}_1, \ldots, \mathsf{x}_n, \mathsf{x}'_1, \ldots, \mathsf{x}'_n$, $k \in \mathbb{Z}$ and $\pm \mathsf{y}$ stands for either y or $-\mathsf{y}$ (same applies for $\pm \mathsf{z}$). Constraints of the form $\pm \mathsf{y} \pm \mathsf{z} \leq k$ are called *octagons* and have been considered in [BGI09]. Note that octagons include constraints of the form $\mathsf{y} \leq \mathsf{z} + k$ or $\mathsf{y} \leq k$ considered in [CJ98]. Unlike the counter systems, in Presburger counter systems with octagonal constraints the transitions do not necessarily lead to functional updates. Here is an example of formula labelling a transition: $\phi = (\mathsf{x}_1 + 1 < \mathsf{x}'_1) \wedge (\mathsf{x}_2 - 3 = \mathsf{x}'_2)$. In [CJ98], Presburger counter systems with octagonal constraints with only constraints of the form $\mathsf{y} \leq \mathsf{z} + k$ or $\mathsf{y} \leq k$ have been studied and a major result established in [CJ98] states that the effect of any loop can be effectively defined in (PA).

Imperfect counter automata. Counter automata are defined as VASS except that we accept also zero-tests on counters as guards. Below, we briefly consider variants of counter automata in which counter values can be decremented without notification (a loss) or counter values can be incremented without notification (a gain) – but not the two possibilities in the same model. A *lossy counter automaton* is a counter automaton such that for all $q \in Q$ and for all $i \in [1,n]$, $q \xrightarrow{\mathrm{dec}(i)} q$ (which allows us to simulate losses). The control state reachability problem for lossy counter automata is decidable and actually lossy counter automata form a subclass of lossy channel systems, see e.g. [Sch02] and the reachability problem for lossy channel systems is decidable [AJ96, FS01]. For instance, they can be used to model lossy channel systems for which the ordering of the messages is not relevant. In that case, each counter can store how many messages of a given type are present in the channel. Lossy counter automata have been introduced in [May03]. Similarly, a *gainy counter automaton* is a counter automaton such that for all $q \in Q$ and for all $i \in [1,n]$, $q \xrightarrow{\mathrm{inc}(i)} q \in \delta$ (which allows us to simulate gains). The control state reachability problem for gainy counter automata can be shown decidable by making a correspondence with reset VASS (VASS in which it is possible to reset counter values) but the problem is nonprimitive recursive, see e.g. [Sch02, Sch10]. Even though Presburger counter systems with imperfect computations are not further discussed in the paper, they form an interesting class of systems related to many other verification problems.

In order to conclude this section, it is worth noting that there exist plenty of other classes of Presburger counter systems for which reachability problems can be solved by using (PA) (see e.g., subclasses of Petri nets). However, since Presburger counter systems are Turing-complete, designing new (tractable) subclasses remains an ongoing process. In the next sections, we focus on presenting proof techniques to solve reachability problems for some of the classes.

3 Loops, Path Schemas and Flatness

3.1 Computing Loop Effects in (PA)

Let $\mathtt{C} = \langle Q, n, \delta \rangle$ be a Presburger counter system. A *path* $\mathtt{p}$ of $\mathtt{C}$ is a finite sequence of transitions from δ corresponding to a path in its control graph. We write *first*($\mathtt{p}$) [resp. *last*($\mathtt{p}$)] to denote the first [resp. last] control state of a path. A *loop* $\mathtt{l}$ is a non-empty path $\mathtt{p}$ such that *first*($\mathtt{p}$) = *last*($\mathtt{p}$) and we write $\mathtt{effect}(\mathtt{l})$ to denote the *effect* of the loop $\mathtt{l}$ defined as below:

$$\{\langle \boldsymbol{x}, \boldsymbol{x'} \rangle \in \mathbb{N}^n \times \mathbb{N}^n : \langle \mathit{first}(\mathtt{l}), \boldsymbol{x} \rangle \xrightarrow{\mathtt{l}} \langle \mathit{last}(\mathtt{l}), \boldsymbol{x'} \rangle\}$$

Similarly, we write $\mathtt{effect}^{<\omega}(\mathtt{l})$ to denote the *repeated effect* of the loop $\mathtt{l}$:

$$\{\langle \boldsymbol{x}, \boldsymbol{x'} \rangle \in \mathbb{N}^n \times \mathbb{N}^n : \langle \mathit{first}(\mathtt{l}), \boldsymbol{x} \rangle \xrightarrow{\mathtt{l}^i} \langle \mathit{last}(\mathtt{l}), \boldsymbol{x'} \rangle,\ i \geq 0\}$$

The reachability problem for loops can be then defined as follows: given a loop $\mathtt{l}$ from a Presburger counter system $\mathtt{C}$ of dimension n and two counter value vectors $\boldsymbol{x_0}$, $\boldsymbol{x_f}$ in $\mathbb{N}^n$, is $\langle \boldsymbol{x_0}, \boldsymbol{x_f} \rangle \in \mathtt{effect}^{<\omega}(\mathtt{l})$? Repeated effect is simply called *acceleration* in [FL02, Section 3].

Note that even though $\mathtt{effect}(\mathtt{l})$ can be defined by a Presburger formula, this does not imply that it is the case for $\mathtt{effect}^{<\omega}(\mathtt{l})$ too. Indeed, if the binary relation R is Presburger set, then this does not imply that its reflexive and transitive closure R^* is a Presburger set too. For instance, if $R = \{\langle \alpha, 2\alpha \rangle \in \mathbb{N}^2 : \alpha \in \mathbb{N}\}$ then $R^* = \{\langle \alpha, 2^\beta \alpha \rangle \in \mathbb{N}^2 : \alpha, \beta \in \mathbb{N}\}$ is not Presburger-definable. By contrast, if $S = \{\langle \alpha, \alpha + 1 \rangle \in \mathbb{N}^2 : \alpha \in \mathbb{N}\}$ then $S^* = \{\langle \alpha, \beta \rangle \in \mathbb{N}^2 : \alpha < \beta,\ \alpha, \beta \in \mathbb{N}\}$ is a Presburger set. The question of deciding whether the reflexive and transitive closure of a Presburger-definable binary relation is Presburger-definable is known to be intimately related to the fact that reachability relations from Presburger counter systems are Presburger-definable, which leads to decidability when effectiveness is guaranteed too. Indeed, consider the following loop with $q_1 = q_k$:

$$q_1 \xrightarrow{\phi_1(\mathsf{x}_1,\ldots,\mathsf{x}'_n)} q_2 \xrightarrow{\phi_2(\mathsf{x}_1,\ldots,\mathsf{x}'_n)} \cdots \xrightarrow{\phi_{k-1}(\mathsf{x}_1,\ldots,\mathsf{x}'_n)} q_{k-1} \xrightarrow{\phi_k(\mathsf{x}_1,\ldots,\mathsf{x}'_n)} q_k.$$

The effect of the loop can be represented by the Presburger formula below:

$$\psi(\bar{\mathsf{x}}, \bar{\mathsf{x}'}) \overset{\text{def}}{=} \exists\ \bar{\mathsf{y}_1}, \ldots, \bar{\mathsf{y}_k}\ \phi_1(\bar{\mathsf{x}}, \bar{\mathsf{y}_1}) \wedge \phi_2(\bar{\mathsf{y}_1}, \bar{\mathsf{y}_2}) \wedge \cdots \wedge \phi_k(\bar{\mathsf{y}_k}, \bar{\mathsf{x}'})$$

where $\bar{\mathsf{x}}$, $\bar{\mathsf{x}'}$, $\bar{\mathsf{y}_1}$, …, $\bar{\mathsf{y}_k}$ are sequences of variables of length n.

In order to decide the reachability problem on the loop, it is essential to represent symbolically $\mathtt{effect}^{<\omega}(\mathtt{l})$. The best we can hope for is that $\mathtt{effect}^{<\omega}(\mathtt{l})$ is a Presburger set. This motivates the definition below.

Definition 3. *Given $R \subseteq \mathbb{N}^{2n}$, we define the* counting iteration of R *as the relation $R_{\mathbf{CI}} \subseteq \mathbb{N}^n \times \mathbb{N} \times \mathbb{N}^n$ such that $\langle \boldsymbol{x}, i, \boldsymbol{y} \rangle \in R_{\mathbf{CI}} \overset{\text{def}}{\Leftrightarrow} \langle \boldsymbol{x}, \boldsymbol{y} \rangle \in R^i$ (i compositions of R). R has a* Presburger counting iteration *if $R_{\mathbf{CI}}$ is a Presburger set.*

If R has a Presburger counting iteration, then there exists a Presburger formula $\varphi(\bar{\mathsf{x}}, \mathsf{z}, \bar{\mathsf{y}})$ such that $[\![\varphi]\!] = R_{\mathbf{CI}}$. Consequently, the relation R^* is Presburger-definable since $[\![\exists\ \mathsf{z}\ \varphi]\!] = R^*$. Observe that $\{\langle \alpha, \alpha + 1 \rangle \in \mathbb{N}^2 : \alpha \in \mathbb{N}\}$ has a Presburger counter iteration witnessed by a Presburger formula of the form $\mathsf{x}' = \mathsf{x} + \mathsf{y}$.

Definition 4 (The property ($\star$) of Presburger counter systems). *A class of Presburger counter systems is said to satisfy the property ($\star$) when, for every loop $\mathtt{l}$, $\mathtt{effect}(\mathtt{l})$ has the Presburger counting iteration and its Presburger formula is computable.*

Note in particular that this means that for every loop $\mathtt{l}$, the set $\{\langle \boldsymbol{x}, i, \boldsymbol{x}' \rangle : \langle \mathit{first}(\mathtt{l}), \boldsymbol{x} \rangle \xrightarrow{\mathtt{l}^i} \langle \mathit{last}(\mathtt{l}), \boldsymbol{x}' \rangle, i \geq 0\}$ is effectively definable by a Presburger formula $\varphi^{\star}_{\mathtt{l}}$ (with $2n + 1$ free variables).

3.2 Finitary Path Schemas

A *path schema* $\mathtt{P}$ is a regular expression built over the alphabet of transitions such that its language represents an overapproximation of the set of labels obtained from finite runs following the transitions of $\mathtt{P}$ (counter values are ignored). This notion has been extensively used since [FO97, Fri00, FL02] and this provides a natural transition since path schemas are made of loops and paths. More precisely, a *finitary path schema* $\mathtt{P}$ is of the form $\mathtt{p}_1 \mathtt{l}_2^* \mathtt{p}_3 \mathtt{l}_4^* \ldots \mathtt{l}_{k-1}^* \mathtt{p}_k$ where (1) $\mathtt{l}_2, \ldots, \mathtt{l}_{k-1}$ are loops and (2) $\mathtt{p}_1 \mathtt{l}_2 \mathtt{p}_3 \mathtt{l}_4 \ldots \mathtt{p}_k$ is a path. The *length* of a path schema, written len($\mathtt{P}$), is defined as the number of letter occurrences in the regular expression defining the path schema (no substructure sharing). Let Lan($\mathtt{P}$) denote the set of finite words in δ^* which belong to the language defined by $\mathtt{P}$. Note that some elements of Lan($\mathtt{P}$) may not correspond to any actual run because of constraints on counter values. Finally, we say that a run ρ starting in a configuration $\langle q_0, \boldsymbol{x_0} \rangle$ *respects* a path schema $\mathtt{P}$ if the sequence of transitions generating ρ belongs to Lan($\mathtt{P}$).

Path schemas are used as a means to encode the structure of a potentially infinite set of runs. That is why, we will pay a special attention to avoid considering distinct path schemas $\mathtt{P}$ and $\mathtt{P}'$ such that Lan($\mathtt{P}$) $\subseteq$ Lan($\mathtt{P}'$). Containment problem for regular expressions is PSPACE-complete but co-NP-complete for regular expressions defining bounded languages, see e.g. [HRS76]. Any set Lan($\mathtt{P}$) defines a bounded language.

That is why, in the following we only consider path schemas such that the loops $\mathtt{l}$ are not multiples of smaller loops (i.e., $\mathtt{l} = (\mathtt{l}')^i$ with $i \geq 2$) and no path $\mathtt{p}$ contains a loop as a factor (which bounds the length of such paths). Such loops are called *simple loops*. In the following, such path schemas are called *good*. It is easy to see that every finite run respects a good path schema.

Lemma 5. *When ($\star$) holds,* $\{\langle \boldsymbol{x}, \boldsymbol{x}' \rangle : \langle q, \boldsymbol{x} \rangle \xrightarrow{*} \langle q', \boldsymbol{x}' \rangle$ respects $\mathtt{P}\}$ *is effectively definable by a Presburger formula $\varphi_{\mathtt{P}}$ (with $2n$ free variables).*

Effective semilinearity boils down to check that the effect of a loop can be characterized by a formula in Presburger arithmetic (or in any decidable fragment of first-order arithmetic). The above result can be then obtained by adequately composing formulae for the loops and for the finite paths.

By way of example, note that the effect of the self-loop $q \xrightarrow{\mathsf{x}'=2\mathsf{x}} q$ is not definable in Presburger arithmetic since $\{2^i : i \geq 0\}$ is not Presburger-definable. By contrast, the effect of the self-loop

$$q \xrightarrow{\mathsf{x}_1'=\mathsf{x}_1+2 \wedge \mathsf{x}_2'=\mathsf{x}_2+3 \wedge \phi(\mathsf{x}_1,\mathsf{x}_2)} q$$

for any Presburger formula $\phi(\mathsf{x}_1, \mathsf{x}_2)$ is Presburger-definable since $\{\langle \boldsymbol{x}, i, \boldsymbol{x}' \rangle : \langle q, \boldsymbol{x} \rangle \xrightarrow{\mathtt{l}^i} \langle q, \boldsymbol{x}' \rangle\}$ can be defined with the formula below:

$$\varphi(\mathsf{x}_1, \mathsf{x}_2, i, \mathsf{x}_1', \mathsf{x}_2') \overset{\text{def}}{=} \mathsf{x}_1' = \mathsf{x}_1+2i \wedge \mathsf{x}_2' = \mathsf{x}_2+3i \wedge \forall\, \mathsf{y}\ (0 \leq \mathsf{y} < i) \Rightarrow \phi(\mathsf{x}_1+2\mathsf{y}, \mathsf{x}_2+3\mathsf{y})$$

Here are concrete classes to apply Lemma 5.

Theorem 6

(I) *Presburger counter systems with octagonal constraints enjoy ($\star$) [BGI09] (see also [CJ98] for a substantial result on a subclass).*
(II) *Counter systems enjoy ($\star$) (folklore result, see e.g. [Fri00, DDS12]).*

An implementation of the transitive closure of octagonal relations is done in the tool FLATA, see e.g., [BGI09].

3.3 Flat Presburger Counter Systems

A Presburger counter system $\mathtt{C}$ is *flat* $\overset{\text{def}}{\Leftrightarrow}$ every control state belongs to at most one simple cycle (i.e., a loop in which each transition occurs at most once). As far as we can judge, the term ‘flat’ in that sense has been introduced in [FO97, CJ98, Fri00]. The Presburger counter system in Figure 1 is flat.

Lemma 7. *Every flat Presburger counter system has a finite number of good path schemas that is at most exponential in its size.*

Of course, this is not the only way to get a finite amount of path schemas, for instance when from an initial configuration, termination is guaranteed but here the finite number of path schemas is structurally guaranteed.

Theorem 8. *Let $\mathcal{C}$ be a class of Presburger counters that enjoys ($\star$). Then, for every flat Presburger counter system from $\mathcal{C}$, the reachability relation* $\mathtt{Reach}_{\mathtt{C}}$ *is Presburger-definable.*

This is at the heart of the decidability results for verifying safety and reachability properties on flat Presburger counter systems from [CJ98, FL02, BIK09] whereas for the verification of temporal properties, it is much more difficult to get sharp complexity characterization, see e.g. [DDS12].

Corollary 9. *Let $\mathcal{C}$ be a class of Presburger counters that enjoys ($\star$). The reachability problem for $\mathcal{C}$ is decidable.*

The corollary can be obtained as follows. Consider the instance $\mathtt{C}$, $\langle q_0, \boldsymbol{x_0}\rangle$ and $\langle q_f, \boldsymbol{x_f}\rangle$. We have seen that we can compute the Presburger formula ϕ that encodes the reachability relation in $\mathtt{C}$. It remains to check satisfiability of the formula $(\bigwedge_{i=1}^{i=n}(\mathsf{x}_i = \boldsymbol{x_0}(i) \wedge \mathsf{x}'_i = \boldsymbol{x_f}(i))) \wedge \phi$ assuming free variables in ϕ are $\mathsf{x}_1, \ldots, \mathsf{x}_n, \mathsf{x}'_1, \ldots, \mathsf{x}'_n$. This can be done thanks to Theorem 1.

3.4 Finite Monoid Property in Affine Presburger Counter Systems

Below, we present a class of affine Presburger counter systems with Presburger-definable loop effects even though the class does not necessarily enjoy the property ($\star$). Given $\mathtt{A} \in \mathbb{Z}^{n\times n}$, let $\mathtt{A}^*$ be the monoid generated from $\mathtt{A}$ with $\mathtt{A}^* = \{\mathtt{A}^i : i \in \mathbb{N}\}$. The identity element is naturally the identity matrix $\mathtt{A}^0 = I$. Given a matrix $\mathtt{A} \in \mathbb{Z}^{n\times n}$, checking whether the monoid generated by $\mathtt{A}$ is finite, is decidable [MS77]. By way of example, with $\mathtt{A} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, we have

$$\mathtt{A}^2 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \quad \mathtt{A}^3 = \begin{pmatrix} 1 & 0 \\ 3 & 1 \end{pmatrix} \ldots \mathtt{A}^m = \begin{pmatrix} 1 & 0 \\ m & 1 \end{pmatrix}$$

So $\mathtt{A}^*$ is not finite. Finiteness of the monoid generated from $\mathtt{A}$ is interesting because of the lemma below.

Lemma 10. *[BW98, Boi99, FL02] Let $R \subseteq \mathbb{N}^n \times \mathbb{N}^n$ be a binary relation of dimension n defined by the triple $\langle \phi_g, \mathtt{A}, \boldsymbol{b}\rangle$ such that $R = \{\langle \boldsymbol{x}, \boldsymbol{x'}\rangle \in \mathbb{N}^{2n} : \boldsymbol{x'} = \mathtt{A}\boldsymbol{x} + \boldsymbol{b}$ and $\boldsymbol{x} \in [\![\phi_g]\!]\}$. If $\mathtt{A}^*$ is finite, then R has a Presburger counting iteration.*

It is worth adding that one can also effectively compute the Presburger formula encoding the relation R^*. A recent work unifying [CJ98, FL02, BGI09, BIL09] by considering all the families of formulae labelling transitions from these works can be found in [BIK09].

A loop in an affine counter system has the *finite monoid property* $\overset{\text{def}}{\Leftrightarrow}$ its corresponding affine update $\langle \phi_g, \mathtt{A}, \boldsymbol{b}\rangle$, possibly obtained by composition of several affine updates, satisfies that $\mathtt{A}^*$ is finite. Let us introduce below the class of admissible counter systems.

Definition 11. *An affine Presburger counter system* $\mathtt{C}$ *is* admissible *iff*

1. *there is at most one transition between two control states (always possible as soon as disjunction is allowed in guards),*
2. *its control graph is flat,*
3. *each simple loop has the finite monoid property.*

The restriction to admissible counter systems mainly takes advantage of Lemma 10 as shown below.

Theorem 12. *[FL02] Let* $\mathtt{C}$ *be an admissible Presburger counter system and* $q, q' \in Q$*. One can effectively compute a Presburger formula* ϕ *such that for every valuation* $\mathfrak{v}$*, we have* $\mathfrak{v} \models \phi$ *iff* $\langle q, \langle \mathfrak{v}(\mathsf{x}_1), \ldots, \mathfrak{v}(\mathsf{x}_n) \rangle \rangle \xrightarrow{*} \langle q', \langle \mathfrak{v}(\mathsf{x}'_1), \ldots, \mathfrak{v}(\mathsf{x}'_n) \rangle \rangle$.

3.5 Flattable Presburger Counter Systems

As observed in [CJ98, FL02, Ler03, BIL09], flatness is very often essential to get effective semilinear reachability sets (but of course this is not a necessary condition, see e.g. [HP79]). However, flat Presburger counter systems are seldom natural in real-life applications. That is why, a relaxed version of flatness has been considered in [FO97, Fri00, LS05, DFGvD10] so that an initialized Presburger counter system $\langle \mathtt{C}, \langle q_0, \boldsymbol{x_0} \rangle \rangle$ is *flattable* whenever there is a partial unfolding of $\langle \mathtt{C}, \langle q_0, \boldsymbol{x_0} \rangle \rangle$ that is flat and has the same reachability set as $\langle \mathtt{C}, \langle q_0, \boldsymbol{x_0} \rangle \rangle$. In that way, reachability questions on $\langle \mathtt{C}, \langle q_0, \boldsymbol{x_0} \rangle \rangle$ can still be decided even in the absence of flatness. $\langle \mathtt{C}, \langle q_0, \boldsymbol{x_0} \rangle \rangle$ is *initially flattable* [LS05] iff there is a *a finite* set of path schemas such that the configurations reachable from $\langle q_0, \boldsymbol{x_0} \rangle$ are those reachable by firing the sequences of transitions from one of those path schemas (not every such sequence leads to a run). For instance, reversal-bounded initialized counter systems are initially flattable [LS05]. The fact that $\langle \mathtt{C}, \langle q_0, \boldsymbol{x_0} \rangle \rangle$ is flattable means that as far as reachability is concerned, a finite set of path schemas captures the full reachability relation. Note that flat counter systems are (structurally) flattable but in general it is non-trivial to compute such a finite set of path schemas, see also Section 5. This problem is also known as the problem of finite *good accelerations* [FL02, Section 5].

4 Verifying Temporal Properties

Reachability problems asks for the existence of runs reaching some configuration or control state in some specific way. Often, it is desirable to check how events are temporally organized along a run and to specify such properties temporal logics have been advocated since [Pnu77]. Furthermore, we wish to include in the logical language the possibility to express directly constraints between variables of the program, whence giving up the standard abstraction made with propositional variables. When the variables are typed, they may be interpreted in some specific domain like integers, real numbers, strings and so on; reasoning in such theories can be performed thanks to satisfiability modulo theories proof techniques, see e.g., [BSST08] and [GNRZ07] in which SMT solvers are used for model-checking infinite-state systems. Hence, a proposition like “x is greater than the next value of y" can be encoded in such extended temporal logics by $\mathtt{x} > \mathtt{X}\mathsf{y}$ but this time the models are sequences of configurations. This means that each position comes with a control state and a valuation for variables. Hence, the basic idea behind the design of the logic Presburger LTL is to refine the language of atomic formulae and to allow the possibility to compare counter values at successive positions of the run of the counter systems. Similar motivations can be found in the introduction of concrete domains in description logics, that are logic-based formalisms for knowledge representation, see e.g. [Lut04].

4.1 Presburger LTL

We define below a version of linear-time temporal logic LTL dedicated to Presburger counter systems in which the atomic formulae are Presburger formulae about counter values, the temporal operators are those of LTL and first-order quantification over natural numbers is allowed, although we shall use it in a restricted way. Similarly, in [MP95], a mixture of first-order logic and LTL is shown sufficient to precisely state verification problems for the class of reactive systems.

We introduce a countable set of *integer variables*, say $\mathrm{VAR}^{\mathrm{P}} = \{\mathsf{y}_1, \mathsf{y}_2, \ldots\}$, for quantification over natural numbers. Elements of $\mathrm{VAR}^{\mathrm{P}}$ are distinct from the *counter variables* in $\mathrm{VAR} = \{\mathsf{x}_1, \mathsf{x}_2, \ldots\}$ that are free variables, only interpreted by the counter values on configurations. We also consider a countably infinite set $\mathtt{Q} = \{q_1, q_2, \ldots\}$ of control state symbols. The Presburger LTL formulae are defined as follows: $\phi ::= \psi \mid q \mid \phi \wedge \phi \mid \neg\phi \mid \mathtt{X}\phi \mid \phi\mathtt{U}\phi \mid \exists\, \mathsf{y}\ \phi$, where ψ is a Presburger formula with free variables in $\mathrm{VAR}^{\mathrm{P}} \cup \mathrm{VAR}$ from the linear fragment of (PA) and $q \in \mathtt{Q}$. The symbols $\mathtt{X}$ and $\mathtt{U}$ are respectively the classical operators next-time and until from LTL.

The models of Presburger LTL formulae are infinite runs from Presburger counter systems whose set of control states is included in the countable set $\mathtt{Q}$. A *model* ρ *of dimension* n *for* Presburger LTL is an element of $(Q \times \mathbb{N}^n)^\omega$ for some finite subset $Q \subseteq \mathtt{Q}$. An *environment* $\mathcal{E}$ is a partial map $\mathrm{VAR}^{\mathrm{P}} \to \mathbb{N}$. The *empty environment* is denoted by $\emptyset$. The satisfiability relation $\models$ is defined as follows between a model ρ of dimension n, a position $i \geq 0$, an environment $\mathcal{E}$ and a formula in which the free variables are among $\mathrm{VAR}^{\mathrm{P}} \cup \{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$.

The relation $\models_{\mathcal{E}}$ is defined on runs $\rho = \langle q_0, \boldsymbol{x_0}\rangle, \ldots, \langle q_k, \boldsymbol{x_k}\rangle, \ldots$ such that:

- $\rho, i \models_{\mathcal{E}} q \overset{\text{def}}{\Leftrightarrow} q = q_i$,
- When ψ is a Presburger formula from the linear fragment with free variables included in $\mathrm{VAR}^{\mathrm{P}} \cup \{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$, we have $\rho, i \models_{\mathcal{E}} \psi \overset{\text{def}}{\Leftrightarrow} \mathfrak{v}_i \models \psi$ in Presburger arithmetic where $\mathfrak{v}_i$ is a conservative extension of $\mathcal{E}$ such that for every $j \in [1, n]$, $\mathfrak{v}_i(\mathsf{x}_j) = \boldsymbol{x_i}(j)$,
- $\rho, i \models_{\mathcal{E}} \neg\phi \overset{\text{def}}{\Leftrightarrow} \rho, i \not\models_{\mathcal{E}} \phi$,
- $\rho, i \models_{\mathcal{E}} \phi_1 \wedge \phi_2 \overset{\text{def}}{\Leftrightarrow} \rho, i \models_{\mathcal{E}} \phi_1$ and $\rho, i \models_{\mathcal{E}} \phi_2$,
- $\rho, i \models_{\mathcal{E}} \mathtt{X}\phi \overset{\text{def}}{\Leftrightarrow} \rho, i+1 \models_{\mathcal{E}} \phi$,
- $\rho, i \models_{\mathcal{E}} \phi_1\mathtt{U}\phi_2 \overset{\text{def}}{\Leftrightarrow}$ there is $j \geq i$ such that $\rho, j \models_{\mathcal{E}} \phi_2$ and $\rho, k \models_{\mathcal{E}} \phi_1$ for all $i \leq k < j$.
- $\rho, i \models_{\mathcal{E}} \exists\, \mathsf{y}\ \phi$ iff there is a natural number $m \in \mathbb{N}$ such that $\rho, i \models_{\mathcal{E}[\mathsf{y} \mapsto m]} \phi$.

As usual, we pose $\mathtt{F}\phi \overset{\text{def}}{=} \top\mathtt{U}\phi$ and $\mathtt{G}\phi \overset{\text{def}}{=} \neg\mathtt{F}\neg\phi$. Semantics with finite runs instead of infinite runs can be defined similarly. A *semi-closed formula* is an Presburger LTL formula such that no integer variable from $\mathrm{VAR}^{\mathrm{p}}$ is free. By construction, the counter variables x_1, …, x_n are always free and are interpreted as the current counter values. In the decision problems defined below, we only consider semi-closed formulae and therefore there is no need to specify an environment in the statements.

For instance, one can express that the first counter strictly increases at every step: $\mathtt{G}\ \exists\ \mathsf{y}\ (\mathsf{y} = \mathsf{x}_1 \wedge \mathtt{X}(\mathsf{x}_1 > \mathsf{y}))$. Similarly, the first counter takes a finite number of values along the run can be expressed by $\exists\ \mathsf{y}\ \mathtt{G}(\mathsf{x}_1 \leq \mathsf{y})$.

Let us start by presenting the *satisfiability problem* for Presburger LTL:

Input: A Presburger LTL semi-closed formula ϕ with free counter variables x_1, …, x_n.
Question: Is there a model $\rho \in (\mathtt{Q} \times \mathbb{N}^n)$ of dimension n such that $\rho, 0 \models_{\emptyset} \phi$?

Observe that for satisfiability checking, it is not necessary that the model is derived from a Presburger counter system. Let us turn to *existential model-checking problem* for Presburger LTL:

Input: A Presburger counter system $\mathtt{C} = \langle Q, n, \delta \rangle$, an initial configuration $\langle q_0, \boldsymbol{x}_0 \rangle$ and a semi-closed formula ϕ in Presburger LTL.
Question: Is there an infinite run ρ starting at $\langle q_0, \boldsymbol{x}_0 \rangle$ such that $\rho, 0 \models_{\emptyset} \phi$?

Temporal logics with Presburger constraints has been developed, for instance, in [BEH95, CC00, BDR03]. Some of them have quite expressive decidable fragments. Undecidability of the existential model-checking problem for Presburger LTL can be shown using the undecidability of the halting problem for Minsky machines, see e.g., [CC00]. Still, using SMT solvers can be done for checking bounded reachability problems, see e.g., [BFM+10].

In the rest of this section, we present fragments of Presburger LTL obtained by restricting first-order quantification over natural numbers. First, let us observe that if we restrict ourselves to formulae in which temporal operators are not in the scope of first-order quantification, then we get a fragment of Presburger LTL that is very similar to plain LTL. Indeed, atomic formulae are arithmetical constraints between counter values and they can be understood as high-level propositional variables; whence the automata-based approach for LTL can be easily adapted to this fragment. In that fragment, the arithmetical constraints are only local and in the construction of Büchi automata, the existence of transitions between states depends on the satisfiability status of Presburger formulae. Below, we provide restrictions in which the temporal operators may occur in the scope of first-order quantification.

Comparing successive counter values. Given a Presburger formula $\psi(\mathsf{z}_1, \ldots, \mathsf{z}_k)$, we shall write $\psi(\mathtt{X}^{i_1}\mathsf{x}_{j_1}, \ldots, \mathtt{X}^{i_k}\mathsf{x}_{j_k})$ to denote the formula below

$$(\exists\ \mathsf{y}_1, \ldots, \mathsf{y}_k\ \mathtt{X}^{i_1}(\mathsf{y}_1 = \mathsf{x}_{j_1}) \wedge \cdots \wedge \mathtt{X}^{i_k}(\mathsf{y}_k = \mathsf{x}_{j_k}) \wedge \psi(\mathsf{y}_1, \ldots, \mathsf{y}_k),$$

where $\mathsf{y}_1, \ldots, \mathsf{y}_k$ are new variables distinct from the free variables that are present in $\psi(\mathsf{z}_1, \ldots, \mathsf{z}_k)$. It is easy to see that $\psi(\mathtt{X}^{i_1}\mathsf{x}_{j_1}, \ldots, \mathtt{X}^{i_k}\mathsf{x}_{j_k})$ is interpreted as the formula $\psi(\mathsf{z}_1, \ldots, \mathsf{z}_k)$ in which each variable z_a takes the value of x_{j_a} at the i_ath next configuration. For instance, $\mathsf{x}_1 = \mathtt{X}\mathsf{x}_2$ specifies that the next value of x_2 is equal to the current value of x_1. Similarly, $\mathtt{G}(\mathsf{x}_1 = \mathtt{X}\mathsf{x}_1)$ states that counter 1 has a constant value along the model. In Section 4.2, we present a simple fragment of Presburger LTL by allowing first-order quantification only for formulae of the form $\psi(\mathtt{X}^{i_1}\mathsf{x}_{j_1}, \ldots, \mathtt{X}^{i_k}\mathsf{x}_{j_k})$ and Presburger formulae (at the atomic level) are quantifier-free too. It is worth observing that we use 'X' as the next-time temporal operator whereas 'Xx' refers to the value of x at the next position.

Freeze operator. In order to verify properties on Presburger counter systems, we want also to be able to compare counter values. For that, it is possible to define the so-called 'freeze operator' with formulae of the form $\downarrow_r^j \phi$ interpreted as $\exists\ \mathsf{y}_r\ (\mathsf{y}_r = \mathsf{x}_j \wedge \phi)$ that store counter values. There are counterpart formulae of the form $\uparrow_r^j$ interpreted as $\mathsf{y}_r = \mathsf{x}_j$ that perform equality tests. Intuitively, the modality $\downarrow_r^j$ is used to store the value of the counter j into the register r; the atomic formula $\uparrow_r^j$ holds true if the value stored in the register r is equal to the current value of the counter j. The formula $\mathtt{G}(\downarrow_1^1\ \mathtt{XG}\neg\uparrow_1^1)$ states that the first counter has distinct values at distinct positions. Freeze operator has been introduced in numerous works, sometimes with different motivations, see e.g. [Hen90, Gor94, Fit02, LP05]. It is also sometimes used implicitly as for the temporal semantics for imperative programs that may use first-order temporal logics, see e.g. [MP92]. For instance, the statement that the program variable x never decreases below its initial value can be expressed by the formula below that uses a form of freeze operator: $\exists \mathsf{y}\ (\mathsf{x} = \mathsf{y})\ \wedge\ \mathtt{G}(\mathsf{x} \geq \mathsf{y})$. Recent results on satisfiability and model-checking problems can be found in [FS09, DLS10].

4.2 The Logic CLTL with Finite Window

As mentioned earlier, we shall define the logic CLTL as a strict fragment of Presburger LTL such that first-order quantification at the level of temporal formulae is restricted to macro formulae of the form $\psi(\mathtt{X}^{i_1}\mathsf{x}_{j_1}, \ldots, \mathtt{X}^{i_k}\mathsf{x}_{j_k})$. Consequently, there is no more quantification over integer variables from $\mathrm{VAR}^{\mathrm{p}}$ and no variable in $\mathrm{VAR}^{\mathrm{p}}$ occurs in CLTL formulae. The logic CLTL has atomic formulae from the linear fragment of (PA) except that variables are replaced by expressions of the form $\mathtt{X}^i\mathsf{x}$ where $\mathsf{x} \in \mathrm{VAR}$ is a variable and $\mathtt{X}^i$ is understood as a sequence of i consecutive symbols X. The expression $\mathtt{X}^i\mathsf{x}$ is interpreted as the value of x at the ith next state. Given a CLTL formula ϕ, we define its X-length $\mathrm{len}(\phi)_{\mathtt{X}}$ as the maximal number i such that an expression of the form $\mathtt{X}^i\mathsf{x}$ occurs in ϕ. Intuitively, the X-length defines the size of a frame/window of consecutive states that can be compared. The models of CLTL are pairs of sequences $\sigma = \langle \sigma_1, \sigma_2 \rangle$ such that $\sigma_1 : \mathbb{N} \to (\mathrm{VAR} \to \mathbb{N})$, $\sigma_2 : \mathbb{N} \to Q$ for a finite subset $Q \subseteq \mathtt{Q}$. The satisfaction relation is defined as for LTL except at the atomic level:

- $\sigma, i \models q$ iff $\sigma_2(i) = q$,
- $\sigma, i \models \psi(\mathtt{X}^{l_1}\mathsf{x}_1, \ldots, \mathtt{X}^{l_n}\mathsf{x}_n)$ iff $\langle \sigma_1(i+l_1)(\mathsf{x}_1), \ldots, \sigma_1(i+l_n)(\mathsf{x}_n)\rangle \in [\![\psi]\!]$.
- $\sigma, i \models \mathtt{X}\phi$ iff $\sigma, i+1 \models \phi$,
- $\sigma, i \models \phi\mathtt{U}\phi'$ iff there is $j \geq i$ such that $\sigma, j \models \phi'$ and for every $i \leq l < j$, we have $\sigma, l \models \phi$.

As usual, a formula $\phi \in$ CLTL is satisfiable whenever there exists a model σ such that $\sigma, 0 \models \phi$. We write CLTL_n^l to denote the restriction of CLTL to formulae with at most n variables and X-length less or equal to l (below the value ω is used for some syntactic resource when there is no restriction). CLTL^0 denote the fragment in which the arithmetical constraints deal only with current counter values. Note that there is a logspace reduction from the satisfiability problem for CLTL to the satisfiability problem for CLTL_ω restricted to formulae of X-length at most 1 (CLTL_ω^1), see e.g. [DLN07]. The proof is done by renaming terms and requires an unbounded amount of variables in CLTL_ω^1. For instance, the expressions $\mathsf{x}_1, \ldots, \mathtt{X}^3\mathsf{x}_1$ are encoded by $\mathtt{G}(\mathsf{x}'' = \mathtt{X}\mathsf{x}' \wedge \mathsf{x}' = \mathtt{X}\mathsf{x} \wedge \mathsf{x} = \mathtt{X}\mathsf{x}_1)$ (assuming that x, x' and x'' are new variables) and each occurrence of $\mathtt{X}\mathsf{x}_1$ [resp. $\mathtt{X}^2\mathsf{x}_1$, $\mathtt{X}^3\mathsf{x}_1$] is replaced by x [resp. x', x'']. For reductions between satisfiability problems, the introduction of new variables is harmless.

The halting problem for Minsky machines can be easily reduced to the satisfiability problem for CLTL or to the existential model-checking problem for CLTL, leading to simple undecidability proofs. In the sequel, we show how to restrict the class of counter systems or the logical language in order to regain decidability.

Given a fragment $\mathfrak{F}$ of (PA) (not necessarily restricted to the linear fragment as for CLTL), we write $\mathrm{CLTL}(\mathfrak{F})$ to denote the variant of CLTL in which atomic formulae built over the quantifier-free linear fragment of (PA) are replaced by formulae from $\mathfrak{F}$ (the definition of the satisfaction relation is update accordingly without significant changes). Similarly, we write $\mathrm{CLTL}_n^l(\mathfrak{F})$ ($n \geq 1$, $l \geq 0$) to denote the restriction of $\mathrm{CLTL}(\mathfrak{F})$ to formulae such that the variables are among $\{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$ and the X-length is bounded by l.

Fragment $\mathfrak{F}_0$ is defined as follows:

$$\mathfrak{F}_0 \ni \phi ::= \mathsf{x}_i < \mathsf{x}_j \;\mid\; \mathsf{x}_i = \mathsf{x}_j \;\mid\; \mathsf{x}_i \leq k$$

where $k \in \mathbb{N}$. Fragment $\mathfrak{F}_1$ is defined as follows:

$$\mathfrak{F}_1 \ni \phi ::= \mathsf{x}_i \sim \mathsf{x}_j + d \;\mid\; \mathsf{x}_i \sim d$$

where $d \in \mathbb{Z}$ and $\sim \in \{<, >, \leq, \geq, =\}$. For instance, $\mathsf{x}_1 = \mathtt{X}^8\mathsf{x}_2 + 1 \in \mathrm{CLTL}_2^8(\mathfrak{F}_1)$ and $\mathtt{XXX}(\mathtt{X}\mathsf{x}_1 \geq 27) \in \mathrm{CLTL}_1^1(\mathfrak{F}_0)$. The logic CLTL defined in [CC00] is precisely $\mathrm{CLTL}_\omega^1(\mathfrak{F}_1)$. By way of example, let us quote a few interesting results.

Theorem 13. *(I) Satisfiability problem for* $\mathrm{CLTL}(\mathfrak{F}_0)$ *is* PSPACE*-complete, see e.g. [DD07, DG08, ST11]. (II) Satisfiability problem for* $\mathrm{CLTL}_1^1(\mathfrak{F}_1)$ *is* PSPACE*-complete [DG09]. (III) Satisfiability problem for* $\mathrm{CLTL}_1^2(\mathfrak{F}_1)$ *or for* $\mathrm{CLTL}_2^1(\mathfrak{F}_1)$ *is undecidable [DG09], see also [CC00].*

4.3 Model-Checking Linear-Time Properties

Flat counter systems. Below, we state several recent results about model-checking problems for subclasses of counter systems for which the complexity is relatively low. We recall that guards in counter systems belong to the linear fragment and the updates are in $\mathbb{Z}^n$.

Theorem 14. *[DDS12] Model-checking flat counter systems with* CLTL^0 *is* NP*-complete (also holds with past-time temporal operators).*

The NP upper bound is obtained as follows given a flat counter system with initial configuration $\langle q_0, \boldsymbol{x_0} \rangle$ and a formula ϕ in CLTL^0:

- Guess a good infinitary path schema $\mathtt{P}$ from $\langle q_0, \boldsymbol{x_0} \rangle$. Infinitary path schemas are of the form $\mathtt{p}_1\mathtt{l}_2^*\mathtt{p}_3\mathtt{l}_4^* \cdots \mathtt{l}_{k-1}^*\mathtt{p}_k\mathtt{l}_k^\omega$.
- Guess an *unfolded path schema* $\mathtt{P}'$ from $\mathtt{P}$ by eliminating disjunctions in guards and counter values (but at the cost of adding new atomic propositions). Unfolding a path schema amounts to copying loops a (polynomial) number of times while adding atomic propositions or constraints in guards to guarantee that each visit in a new loop satisfies the same guards.
- Build an existential Presburger formula that encodes all the runs respecting $\mathtt{P}'$ from $\langle q_0, \boldsymbol{x_0} \rangle$ (all the quantified variables are loop counters).
- Guess a run respecting $\mathtt{P}'$ and check whether it satisfies ϕ symbolically. Each loop may be visited an exponential number of times, but a stuttering theorem allows the symbolic model-checking algorithm to perform efficiently.

Efficient solvers for quantifier-free (PA) are required to make feasible the above algorithm. By contrast, we get a little higher complexity with linear μ-calculus or first-order logic (FOL).

Theorem 15. *[DDS13] Model-checking flat counter systems with linear μ-calculus or with FOL both with arithmetical constraints is* PSPACE*-complete.*

It is unclear what are the counterpart results for flat Presburger counter systems with octagonal constraints.

LTL on VASS. Structural restrictions seem more efficient to reduce the computational complexity of temporal model-checking. By way of comparison, model-checking vector addition systems with states with linear μ-linear calculus (without arithmetical constraints) is already EXPSPACE-complete [Hab97].

Let us present a fragment of Presburger LTL introduced in [Jan90] such that the atomic formulae are either control states or atomic formulae of the form $\mathsf{x}_i \geq k$ or $\neg(\mathsf{x}_i \geq k)$ with $k \in \mathbb{N}$. The temporal logic with fairness TLF is defined as a logic on VASS for which formulae are defined by the grammar $q \mid \mathsf{x}_i \geq k \mid \neg(\mathsf{x}_i \geq k) \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathtt{GF}\phi$, where $q \in \mathtt{Q}$ and $k \in \mathbb{N}$. Observe that TLF formulae are not closed under negations and the temporal properties are intersection or union of fairness conditions. Decidability of (existential) model-checking problem for TLF restricted to VASS is established in [Jan90] by reduction into the reachability problem for VASS. Fairness conditions on VASS can be also found in [GS92]. Moreover, it is worth noting that the operator $\mathtt{F}$ cannot be expressed in TLF, otherwise undecidability would hold. Indeed, in [HR89] a linear-time temporal logic (on Petri nets) is shown undecidable with the temporal operator $\mathtt{F}$, Boolean connectives and atomic formulae of the form $\mathsf{x}_i \geq k$ and "transition t is the next one in the run." Finally, other logical formalisms interpreted on VASS runs can be found in [Esp94, AH11, BS11] where complexity/decidability results are established.

Bounding the number of reversals. Results about model-checking reversal-bounded counter systems with LTL equipped with arithmetical constraints can be found in [BD11, HL11]. Below, we recall the definition for the reversal-bounded model-checking problem (RBMC). Its peculiarity is that the input initialized counter systems are not necessarily reversal-bounded but the input contains an explicit bound r about the maximal number of reversals within a run. Moreover, given a formula ϕ in CLTL, we write $\mathtt{T}_\phi$ to denote the terms of the form $\mathtt{t} \sim k$ occurring in it. The problem RBMC is defined as follows:

Input: a counter system $\mathtt{C}$, an initial configuration $\langle q_0, \boldsymbol{x}_0 \rangle$, a formula CLTL ϕ (with atomic formulae of the form $\mathtt{t} \sim k$), a bound $r \in \mathbb{N}$ (in binary),
Question: Is there an infinite run ρ from $\langle q, \boldsymbol{x} \rangle$ such that $\rho, 0 \models \phi$ and ρ is r-$\mathtt{T}$-reversal-bounded with $\mathtt{T} = \mathtt{T}_{\mathtt{C}} \cup \mathtt{T}_\phi$?

The computational complexity for RBMC can be precisely characterized; the upper bound can be obtained by a refined analysis on runs, see e.g. [GI81, BD11].

Theorem 16. *[BD11, HL11] RBMC is* NExpTime*-complete.*

Actually, one can also establish that global model-checking is possible for RBMC [BD11], i.e., the set of initial configurations for which there is a reversal-bounded run satisfying a given temporal formula from CLTL is effectively Presburger-definable.

4.4 A Quick Look at a Branching-Time Extension

The logic Presburger LTL is interpreted on linear runs but it is possible to extend it to its CTL* version by interpreting the formulae on the underlying transition systems of the Presburger counter systems and by adding quantifications over paths, see e.g. [BG06, DFGvD10]. The formulae for Presburger CTL* are defined as follows: $\phi ::= \psi \mid q \mid \phi \wedge \phi \mid \neg\phi \mid \mathtt{X}\phi \mid \phi\mathtt{U}\phi \mid \mathtt{A}\ \phi \mid \exists\ \mathsf{y}\ \phi$ where ψ is a Presburger formula with free variables included in $\mathrm{VAR}^{\mathrm{P}} \cup \mathrm{VAR}$ from the linear fragment and $q \in \mathtt{Q}$. Semantics for Presburger CTL* is provided via models that are transition systems obtained from Presburger counter systems. Again, the satisfaction relation $\models$ is parameterized by an *environment* $\mathcal{E}$. Given a Presburger counter system $\mathtt{C} = \langle Q, n, \delta \rangle$ with transition system $\mathfrak{T}(\mathtt{C}) = \langle \mathfrak{S}, \rightarrow \rangle$, the satisfaction relation $\models_{\mathcal{E}}$ is defined at position i of the run as for Presburger LTL except for quantifications over paths: $\rho, i \models_{\mathcal{E}} \mathtt{A}\ \ \phi \overset{\text{def}}{\Leftrightarrow}$ for all infinite runs ρ' starting at configuration $\rho(i)$, we have $\rho', 0 \models_{\mathcal{E}} \phi$. First-order quantification over counter values allows us to state many interesting properties in Presburger CTL* such as determinism (for all the configurations reachable from the initial configuration, there is at most one successor configuration):

$$\mathtt{A}\ \mathtt{G}(\bigwedge_{i \in [1,n]} \neg \exists \mathsf{y}(\mathtt{E}\ \mathtt{X}(\mathsf{x}_i = \mathsf{y}) \wedge \mathtt{E}\ \mathtt{X}(\mathsf{x}_i \neq \mathsf{y}))) \wedge (\bigwedge_{q \in Q} \neg(\mathtt{E}\ \mathtt{X}q \wedge \mathtt{E}\ \mathtt{X}\neg q)).$$

Similarly, boundedness (the set of configurations reachable from the initial configuration is finite) can be stated with $\exists \mathsf{y}, \mathsf{y}'\ \mathtt{A}\ \mathtt{G} \bigwedge_{i \in [1,n]} \mathsf{y} \leq \mathsf{x}_i \leq \mathsf{y}'$.

Model-checking problem for Presburger CTL* is defined as follows: given a Presburger counter system $\mathtt{C}$ with transition system $\mathfrak{T}(\mathtt{C}) = \langle \mathfrak{S}, \rightarrow \rangle$, an initial configuration $\langle q_0, \boldsymbol{x}_0 \rangle$, and a semi-closed formula ϕ from Presburger CTL*, determine whether for every run ρ from $\langle q_0, \boldsymbol{x}_0 \rangle$, we have $\rho, 0 \models \phi$.

Theorem 17. *[DFGvD10] Model-checking admissible Presburger counter systems with Presburger CTL* is decidable.*

The proof provides a reduction into satisfiability in (PA) by encoding the runs by tuples of natural numbers. Indeed, every admissible Presburger counter system is flat and therefore it has a finite amount of good path schemas. Runs respecting a path schema can be encoded as tuples of natural numbers by counting how many times the loops are visited. Temporal formulae are then encoded by internalizing the semantics into (PA) itself. Other decidability and complexity results can be found in [BP12, CKL13]. Nevertheless, it remains open whether modal μ-calculus (with atomic arithmetical constraints on counter values) can be shown decidable on admissible Presburger counter systems.

5 Path Schema Enumeration

In this section, we explain the interest of designing algorithms for the enumeration of finitary path schemas and how to prune the search space by subsumption. We present a preliminary version of an algorithm for enumerating path schemas. More details and developments will be provided in a forthcoming paper. Only finitary path schemas are discussed in this section but infinitary ones could be generated in a similar way. Moreover, we assume that we are dealing with a class of Presburger counter systems satisfying the property ($\star$), recalling Definition 4 on page 129, such as the class of counter systems where the guards are Boolean combinations of linear constraints and the updates in $\mathbb{Z}^n$ are those from VASS.

5.1 Why Path Schema Enumeration?

As is well-known, Presburger counter systems are Turing-complete and it is undecidable to check the existence of a run satisfying a given property (even for very basic ones). However, approximating the Presburger counter systems by looking at a subclass of runs provides a means to produce answers in some cases. For example, a finite set of path schemas is a simple way to represent a (potentially) infinite set of runs. Being able to generate path schemas in a structured and controlled fashion while using structural properties of the control graph as well as arithmetical constraints of counter values will be helpful to test the existence of runs satisfying some property.

A wish list for generating path schemas. Even though it is not difficult to generate path schemas in a fair and complete way by tracing the transitions, the details of the enumeration are quite important but often underestimated, see e.g.. [BFLS05, DFGvD10] (see some exception in [Ler03]). First, we want an enumeration strategy that is efficient in practice. Previous work has left open the question of efficient enumeration of path schemas, as the results have been of a theoretical nature, and path schemas didn't have to be enumerated explicitly. Second, we want an enumeration strategy that will find a finite set of path schemas that fully captures the behavior of a Presburger counter system, if possible. As noted earlier, some classes of Presburger counter systems are known to have a finite number of good path schemas. For flat Presburger counter systems, the number of good path schemas is exponential in the size of the control graph whereas for flattable initialized Presburger counter systems, the number of path schemas is finite but with no specific bound on this number. Finally, we want an enumeration strategy in which we have a clear way of detecting whether we have enumerated sufficiently many path schemas to capture the behavior of the Presburger counter system.

Enumerating path schemas can also be viewed as a way to underapproximate the set of runs; this is similar to a standard approach to consider subclasses of runs by bounding some features and to search for 'bounded runs' that may satisfy a desirable or undesirable property. Examples include reversal-bounded counter machines (which have a bound on the number of reversals) [HR87, BD11, HL11], context-bounded model-checking (where there is a bound on the number of context switches) [QR05], and of course bounded model-checking (BMC) (where there is a bound on the distance of the reached positions), see e.g. [BCC$^+$03].

5.2 Pruning the Search Space: Path Schema Subsumption

Let $\mathtt{C} = \langle Q, n, \delta \rangle$ be a Presburger counter system and $\mathtt{P}_1, \ldots, \mathtt{P}_N, \mathtt{P}$ be finitary path schemas such that $\mathit{first}(\mathtt{P}_1) = \cdots = \mathit{first}(\mathtt{P}_N) = \mathit{first}(\mathtt{P})$ and $\mathit{last}(\mathtt{P}_1) = \cdots = \mathit{last}(\mathtt{P}_N) = \mathit{last}(\mathtt{P})$, i.e., all the path schemas start and end by the same control states. One can think of $\mathtt{P}_1$, ..., $\mathtt{P}_N$ as the path schemas already in our database whereas $\mathtt{P}$ is a new path schema for which we have to decide whether we keep it or not. Such a path schema $\mathtt{P}$ must be consistent, i.e., there exists a finite run that respects it. The path schema $\mathtt{P}$ is *consistent* w.r.t. the initial condition $\phi_{\mathrm{init}}(\mathtt{y}_1, \ldots, \mathtt{y}_n)$ iff the formula $\exists\ \mathtt{x}_1, \ldots, \mathtt{x}_n\ \exists\ \mathtt{x}'_1, \ldots, \mathtt{x}'_n\ \phi_{\mathrm{init}}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \wedge \varphi_{\mathtt{P}}(\mathtt{x}_1, \ldots, \mathtt{x}_n, \mathtt{x}'_1, \ldots, \mathtt{x}'_n)$ is valid. Then, comes subsumption. The set $\{\mathtt{P}_1, \ldots, \mathtt{P}_N\}$ *subsumes* $\mathtt{P}$ *w.r.t. the initial condition* $\phi_{\mathrm{init}}(\mathtt{y}_1, \ldots, \mathtt{y}_n)$ (and with respect to reachability) $\overset{\text{def}}{\Leftrightarrow}$ the formula below is valid:

$$\forall\, \mathsf{x}_1,\ldots,\mathsf{x}_n\ \forall\, \mathsf{x}'_1,\ldots,\mathsf{x}'_n\ (\phi_{\mathrm{init}}(\mathsf{x}_1,\ldots,\mathsf{x}_n) \wedge \varphi_{\mathtt{P}}(\mathsf{x}_1,\ldots,\mathsf{x}_n,\mathsf{x}'_1,\ldots,\mathsf{x}'_n)) \Rightarrow$$

$$\bigvee_{i\in[1,N]} \exists\, \mathsf{z}_1,\ldots,\mathsf{z}_n\ \phi_{\mathrm{init}}(\mathsf{z}_1,\ldots,\mathsf{z}_n) \wedge \varphi_{\mathtt{P}_i}(\mathsf{z}_1,\ldots,\mathsf{z}_n,\mathsf{x}'_1,\ldots,\mathsf{x}'_n)$$

For the class of counter systems, the consistency problem is NP-complete, and the subsumption problem can be expressed in the fragment of (PA) with at most one quantifier alternation. The above subsumption notion can be also formulated as follows. A path schema $\mathtt{P}$ enriched with $\phi_{\mathrm{init}}(\mathsf{y}_1,\ldots,\mathsf{y}_n)$ defines a set of finite runs such that the initial counter values satisfy $\phi_{\mathrm{init}}(\mathsf{y}_1,\ldots,\mathsf{y}_n)$ and the run respects $\mathtt{P}$. Moreover, such a pair defines a set of counter values—those that have been reached at the end of the runs, say $[\mathtt{P}]_{\phi_{\mathrm{init}}} \overset{\mathrm{def}}{=} \{\boldsymbol{x_f} : \langle q_0, \boldsymbol{x_0}\rangle \xrightarrow{*} \langle q_f, \boldsymbol{x_f}\rangle$ respects $\mathtt{P}$ and $\phi_{\mathrm{init}}(\boldsymbol{x_0})\}$. Counter values are therefore extracted from runs. Now, subsumption can be formulated as follows: $[\mathtt{P}]_{\phi_{\mathrm{init}}} \subseteq [\mathtt{P}_1]_{\phi_{\mathrm{init}}} \cup \cdots \cup [\mathtt{P}_N]_{\phi_{\mathrm{init}}}$. There exist other means to extract witness counter values from runs. A *pattern* ϕ_{pat} is a formula from Presburger LTL without first-order quantification (see Section 4) and with free occurrences of the integer variables y_1, ... y_α that are therefore interpreted rigidly. By way of example, we consider the version of Presburger LTL on finite runs. Let us define the set of tuples $[\mathtt{P}]_{\phi_{\mathrm{pat}},\phi_{\mathrm{init}}}$ obtained by extracting the parameter values from runs respecting $\mathtt{P}$ and whose initial configuration satisfies ϕ_{init} (see the semantics in Section 4):

$$[\mathtt{P}]_{\phi_{\mathrm{pat}},\phi_{\mathrm{init}}} \overset{\mathrm{def}}{=} \{\mathcal{E} : \rho = \langle q_0, \boldsymbol{x_0}\rangle \xrightarrow{*} \langle q_f, \boldsymbol{x_f}\rangle \text{ respects } \mathtt{P},\quad \phi_{\mathrm{init}}(\boldsymbol{x_0})\ \&\ \rho, 0 \models_{\mathcal{E}} \phi_{\mathrm{pat}}\}$$

Note that $[\mathtt{P}]_{\phi_{\mathrm{init}}}$ above corresponds to $[\mathtt{P}]_{\phi_{\mathrm{pat}},\phi_{\mathrm{init}}}$ with

$$\phi_{\mathrm{pat}} \overset{\mathrm{def}}{=} \mathtt{F}(\mathsf{x}_1 = \mathsf{y}_1 \wedge \cdots \wedge \mathsf{x}_n = \mathsf{y}_n \wedge \neg \mathtt{X}\top)$$

Let us define the generalized path schema subsumption problem: $\{\mathtt{P}_1,\ldots,\mathtt{P}_N\}$ *subsumes* $\mathtt{P}$ *with respect to the initial condition* $\phi_{\mathrm{init}}(\mathsf{y}_1,\ldots,\mathsf{y}_n)$ *and the property/pattern* ϕ_{pat} $\overset{\mathrm{def}}{\Leftrightarrow}$ $[\mathtt{P}]_{\phi_{\mathrm{pat}},\phi_{\mathrm{init}}} \subseteq [\mathtt{P}_1]_{\phi_{\mathrm{pat}},\phi_{\mathrm{init}}} \cup \cdots \cup [\mathtt{P}_N]_{\phi_{\mathrm{pat}},\phi_{\mathrm{init}}}$ (of course, a Presburger counter system is also part of the instance).

Lemma 18. *For any class of Presburger counter systems satisfying ($\star$), there is a reduction from the generalized path schema subsumption problem to the validity problem for (PA).*

The proof consists in encoding the runs satisfying a path schema by tuples and then to use the standard translation from LTL to first-order logic. The initial condition ϕ_{init} and atomic formulae in PLTL formulae are already Presburger formulae, so do not require special treatment in the translation process.

5.3 How to Deal with Quantifiers

Note that Presburger formulae built to perform subsumption tests contain quantifiers. Most well-known Satisfiability Modulo Theories (SMT) solvers can deal with quantifier-free formulae, also known as linear arithmetic (LIA). For instance, this includes Z3 [dMB08], CVC4 [BCD+11], and Alt-Ergo [Con12], to cite a few of them; see also Pugh's Omega test [Pug92].

However, as observed earlier, dealing with quantifiers is usually a difficult task for SMT solvers. Fortunately, quantifiers can be eliminated but this may be expensive computationally. Cooper's elimination procedure [Coo72], when considering $\exists\ \mathsf{x}\ \psi$ with quantifier-free ψ, does not assume that ψ is in disjunctive normal form (a disjunction of conjuncts, with conjuncts made of literals). This is a remarkable difference with the original algorithm presented in [Pre29]. Indeed, transforming a propositional formula into an equivalent formula in disjunctive normal form may cause an exponential blow-up. A more advanced improvement of Cooper's procedure can be found in [RL78]; recent developments propose a lazy approach to quantifier elimination [Mon10].

5.4 An Algorithm that Builds Cycle Schemas and Path Schemas

In this section, we present an algorithm to generate path schemas from a Presburger counter system. It proceeds by building path schemas of larger and larger sizes. An outer loop ensures that all path schemas of some constant size $k-1$ have been built before the generation of path schemas of size k is attempted. The algorithm is inherently iterative, and its first k iterations enumerate all path schemas of size less than or equal to k.

Path schemas are generated by building upon smaller path schemas. Given a path schema of size $k-1$, the algorithm extends it by adding a transition; the result is a new path schema of size k. Path schemas may also be extended by cycles, and for this, the algorithm detects and maintains *cycle schemas* (see below). These cycle schemas are detected by using smaller path schemas.

Preliminary definitions: cycle schemas and suffixes. A *cycle schema* L is a path schema starting and ending by the same control state. The set of cycles generated by a cycle schema L is precisely $\mathrm{Lan}(L)$. We write $\mathrm{Lan}^{\circlearrowright}(L)$ to denote the smallest set of paths such that $\mathrm{Lan}(L) \subseteq \mathrm{Lan}^{\circlearrowright}(L)$ and if $t_1 \cdots t_\alpha \in \mathrm{Lan}^{\circlearrowright}(L)$, then $t_2 \cdots t_\alpha t_1 \in \mathrm{Lan}^{\circlearrowright}(L)$. The set $\mathrm{Lan}^{\circlearrowright}(L)$ can be also obtained from $\mathrm{Lan}(L)$ by considering all possible rotations of loops.

Path schemas can be concatenated assuming that constraints on control states are respected. Let $\mathtt{P} = \mathtt{P}_1 \cdot \mathtt{P}_2$ be a path schema obtained by concatenating two path schemas such that (1) $\mathtt{P}_2$ starts by q' and is of length at least one, (2) $\mathtt{P}_2$ ends by q, (3) there is a transition $t = q \xrightarrow{\phi} q'$. Obviously, $\mathtt{P}_2 \cdot t$ is a cycle schema. $\mathtt{P}_2$ is called a *suffix* of $\mathtt{P}$. Similarly, let $\mathtt{P} = \mathtt{P}_1 \cdot (\mathtt{l})^* \cdot \mathtt{P}_2$ be a path schema with $\mathtt{l} = \mathtt{p}_1 \cdot \mathtt{p}_2$, such that (1) $\mathtt{p}_2$ starts by q', (2) $\mathtt{P}_2$ ends by q, (3) there is a transition $t = q \xrightarrow{\phi} q'$. Obviously, $\mathtt{p}_2 \cdot (\mathtt{l})^* \cdot \mathtt{P}_2 \cdot t$ is another cycle schema. $\mathtt{p}_2 \cdot (\mathtt{l})^* \cdot \mathtt{P}_2$ is called an *augmented suffix* of $\mathtt{P}$. By definition, an augmented suffix is any suffix obtained in one of the two above-mentioned ways. A *simple suffix* is a suffix without a loop, i.e., a non-empty sequence of transitions being the suffix of a path schema.

ps-complexity. A path $\mathtt{p} \in \delta^*$ has *ps-complexity* $k \overset{\mathrm{def}}{\Leftrightarrow}$ there is a path schema $\mathtt{P}$ of length k such that $\mathtt{p} \in \mathrm{Lan}(\mathtt{P})$ and no path schema $\mathtt{P}'$ of strictly smaller size verifies $\mathtt{p} \in \mathrm{Lan}(\mathtt{P}')$. In a sense, the ps-complexity of a path measures how concisely the path can be represented with the help of path schemas generated from the Presburger counter system. Similarly, a run ρ has *ps-complexity* $k \overset{\text{def}}{\Leftrightarrow}$ there is a path schema $\mathtt{P}$ of length k such that ρ respects $\mathtt{P}$ and for no path schema $\mathtt{P}'$ of strictly smaller size, ρ respects $\mathtt{P}'$. The *relative length* of a loop $\mathtt{l}$ with respect to control state q_0 is equal to the length of $\mathtt{l}$ plus the minimal distance between the initial control state q_0 and a control state occurring in $\mathtt{l}$ (can be infinite, can be equal to the length of $\mathtt{l}$ if q_0 occurs in $\mathtt{l}$). Again, no constraints about counter values are involved at this stage.

Subsumption test. Our algorithm is parameterized by a subsumption test. When a path schema is subsumed by the set of path schemas previously discovered, it is not itself enumerated. This leads to less redundant results, and to less work being done by the algorithm at larger k. It also leads to an easy termination test: during a level k, if no new path schemas are enumerated, then the algorithm has already enumerated a finite set of path schemas that "capture" the behavior of the system (w.r.t. the subsumption test).

The algorithm. Below, we present the algorithm in which $PS(k)$ is the set of path schemas of size k discovered. $CS(k)$ is the set of cycle schemas discovered with relative length k. We write $PS^{+}(k) \overset{\text{def}}{=} \bigcup_{k' \leq k} PS(k')$ and $CS^{+}(k) \overset{\text{def}}{=} \bigcup_{k' \leq k} CS(k')$. The input is a Presburger counter system $\mathtt{C}$, with initial state q_0 and initial condition $\phi_{\text{init}}(\mathsf{x}_1, \ldots, \mathsf{x}_n)$. The input contains a path schema eligibility test (called *test*) so that $PS(k) \xleftarrow{test} \mathtt{P}$ is a shorthand for: `if` $test(\phi_{\text{init}}, \mathtt{P}, PS^{+}(k))$ `then` $PS(k) \longleftarrow PS(k) \cup \{\mathtt{P}\}$. When the test returns true, a new path schema is included in $PS(k)$ (typically by performing a subsumption check). The output of the algorithm is $PS^{+}(k)$.

```
1. $PS(0)$ is initialized to the empty path schema starting at control state $q_0$
2. $k \longleftarrow 1$
3. while $PS^{+}(k-1) \neq \emptyset$ do
   (a) for each $\mathtt{P} \in PS^{+}(k-1)$ and $t \in \delta$ s.t. $first(t) = last(\mathtt{P})$ do
       { look for path schemas ending with a transition }
        i. if there exists no simple suffix $S$ of $\mathtt{P} \cdot t$ s.t. $first(S) = last(S)$ then
           $PS(k) \xleftarrow{test} \mathtt{P} \cdot t$ { add path schema $\mathtt{P} \cdot t$ to level $k$ }
       ii. for each augmented suffix $S$ of $\mathtt{P}$ s.t. $first(S \cdot t) = last(S \cdot t)$ do
           { add cycle schema $S \cdot t$ to level $\mathrm{len}(S \cdot t)$ }
           $CS(\mathrm{len}(S \cdot t)) \longleftarrow CS(\mathrm{len}(S \cdot t)) \cup \{S \cdot t\}$
   (b) { add path schemas ending with a cycle }
       for each $L \in CS^{+}(k)$ and prime cycle $\mathtt{l} \in \mathrm{Lan}^{\circlearrowleft}(L)$ s.t. $\mathrm{len}(\mathtt{l}) \leq k$ do
           for each $\mathtt{P} \in PS(k - \mathrm{len}(\mathtt{l}))$ s.t. $last(\mathtt{P}) = first(\mathtt{l})$ do
               $PS(k) \xleftarrow{test} \mathtt{P} \cdot \mathtt{l}$ { add $\mathtt{P} \cdot \mathtt{l}$ to level $k$ }
   (c) $k \longleftarrow k + 1$; endwhile
4. return $PS^{+}(k)$
```

Properties of the algorithm. The algorithm has several nice properties such as being parameterized by an eligibility test and it produces only good path schemas (see line 3(a)(i)). It is natural to wonder about the purpose of the eligibility test. Whenever a new path schema is built by the algorithm, we do not systematically insert it in the working set of path schemas (represented by $PS(k)$ or $PS^{+}(k)$). Indeed, it may happen that there is no run that respects it when starting by the initial control state q_0 and when satisfying the initial constraint on counter values. In that case, there is no point to include it in the working set of path schemas. When path schemas are generated with the purpose to abstract a potentially infinite set of runs, only consistent path schemas are inserted. Similarly, a new path schema may be subsumed by the working set of path schemas and if the property to be checked on runs can be safely pruned, such a new path schema can be discarded. The eligibility test allows to parameterize the algorithm by any kind of Boolean function to test whether a new path schema can be inserted or not. On the other hand, it might be useful to enumerate path schemas regardless the arithmetical constraints on counter values, which corresponds to consider the algorithm when the test always returns true. Consequently, the eligibility test provides a means to eliminate new path schemas depending on the purpose of the path schema generation.

Theorem 19 below states that the algorithm generates all the cycle schemas and path schemas when constraints on counter values are ignored. A nice way to ignore such values is to assume that the test returns always true, which amounts also to view the counter system as a standard labelled transition system.

Theorem 19 (Completeness). *Consider the algorithm in which the main test returns true. After completing the kth step of the main while loop:*

(†) *For every loop* $\mathtt{l}$ *of relative length at most* k*, there is a cycle schema* $L \in CS^{+}(k)$ *such that* $\mathtt{l} \in \mathrm{Lan}^{\circlearrowleft}(L)$.

(††) *For every path* $\mathtt{p}$ *starting at control state* q_0 *of ps-complexity at most* k*, there is a path schema* $\mathtt{P} \in PS^{+}(k)$ *such that* $\mathtt{p} \in \mathrm{Lan}(\mathtt{P})$.

With subsumption on counter values, a complete version of the algorithm can be obtained if cycles are generated independently of cycle schemas. At the time of writing this paper, we have designed such an algorithm, based on the one presented above. If cycle schemas are used to generate cycles during the course of the algorithm, then the enumeration procedure is known to be incomplete in the sense of case (††) in Theorem 19; that is, some path schemas may be missed at step k that are necessary to describe a path of ps-complexity k. However, this does not prevent us from using this algorithm for certain applications where completeness is less important, as useful path schemas might still be generated. Implementation and tests will be part of future work.

6 Conclusion

In this paper, we have recalled several classes of Presburger counter systems for which reachability sets are computable Presburger sets. Though this is a desirable property to provide decision procedures on such machines, it is not sufficient if model-checking temporal properties are required; indeed, we may need to specify how intermediate configurations occur (see e.g. [DDS12]). For instance, the exact complexity of model-checking temporal properties for flat admissible Presburger counter systems is still open.

We have recalled several results from the literature and we emphasize that the generation of path schemas is a key problem for formal verification of Presburger counter systems. This is not really new (see e.g. [FO97, Boi99, Ler03, LS05]) but it is becoming an important issue, at least as important as the design of optimal decision procedures as far as worst-case complexity is concerned. The paper has been designed to put some light on this problem. However, an efficient generation of path schemas means that redundant path schemas should be eliminated as early as possible in the enumeration process. A comparison with the algorithm for acceleration technique in FAST [Ler03] or LASH [Boi99] will be in order.

We have introduced the notion of subsumption to take care of redundancy and again subsumption can be checked by testing the satisfiability of a quantified Presburger formula. This is a real challenge to deal with such quantified formulae in the framework of path schemas enumeration since most SMT solvers do not behave so nicely with quantified formulae, see e.g. [dMB08, BCD$^+$11]. Part of our future work is dedicated to design a path schema generation algorithm that invokes SMT solvers for quantified Presburger formulae.

Acknowledgements. The second author thanks the colleagues involved in fruitful collaborations about Presburger counter systems along the years; including members of the ANR Project REACHARD, R. Lazić (Warwick University), A. Dhar and A. Sangnier (LIAFA), and members of the ACSys group at NYU.

References

[AH11] Atig, M.F., Habermehl, P.: On Yen's path logic for Petri nets. IJFCS 22(4), 783–799 (2011)

[AJ96] Abdulla, P., Jonsson, B.: Verifying programs with unreliable channels. I & C 127(2), 91–101 (1996)

[BBH$^+$06] Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with Lists Are Counter Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)

[BCC$^+$03] Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)

[BCD$^+$11] 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)

[BD11] Bersani, M.M., Demri, S.: The complexity of reversal-bounded model-checking. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS (LNAI), vol. 6989, pp. 71–86. Springer, Heidelberg (2011)

[BDR03] Bruyère, V., Dall'Olio, E., Raskin, J.-F.: Durations, parametric model-checking in timed automata with Presburger arithmetic. In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 687–698. Springer, Heidelberg (2003)

[BEH95] Bouajjani, A., Echahed, R., Habermehl, P.: On the verification problem of nonregular properties for nonregular processes. In: LICS 1995, pp. 123–133 (1995)

[Ber80] Berman, L.: The complexity of logical theories. TCS 11, 71–78 (1980)

[BFLS05] Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 474–488. Springer, Heidelberg (2005)

[BFM$^+$10] Bersani, M.M., Frigeri, A., Morzenti, A., Pradella, M., Rossi, M., San Pietro, P.: Bounded reachability for temporal logic over constraint systems. In: TIME 2010, pp. 43–50. IEEE (2010)

[BG06] Bozzelli, L., Gascon, R.: Branching-time temporal logic extended with qualitative Presburger constraints. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 197–211. Springer, Heidelberg (2006)

[BGI09] Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 337–351. Springer, Heidelberg (2009)

[BIK09] Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010)

[BIL09] Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. FI 91(2), 275–303 (2009)

[BJW01] Boigelot, B., Jodogne, S., Wolper, P.: On the use of weak automata for deciding linear arithmetic with integer and real variables. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 611–625. Springer, Heidelberg (2001)

[BKR96] Biehl, M., Klarlund, N., Rauhe, T.: MONA: Decidable arithmetic in practice. In: Jonsson, B., Parrow, J. (eds.) FTRTFT 1996. LNCS, vol. 1135, pp. 459–462. Springer, Heidelberg (1996)

[BL10] Bojańczyk, M., Lasota, S.: An extension of data automata that captures XPath. In: LICS 2010, pp. 243–252. IEEE (2010)

[Boi99] Boigelot, B.: Symbolic methods for exploring infinite state spaces. PhD thesis, Université de Liège (1999)

[BP12] Bozzelli, L., Pinchinat, S.: Verification of gap-order constraint abstractions of counter systems. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 88–103. Springer, Heidelberg (2012)

[BS11] Blockelet, M., Schmitz, S.: Model checking coverability graphs of vector addition systems. In: Murlak, F., Sankowski, P. (eds.) MFCS 2011. LNCS, vol. 6907, pp. 108–119. Springer, Heidelberg (2011)

[BSST08] Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability Modulo Theories. Frontiers in Artificial Intelligence and Applications, vol. 185, ch. 26, pp. 825–885. IOS Press (2008)

[BST12] Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: Version 2.0 (September 2012), `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf`

[BW94] Boigelot, B., Wolper, P.: Verification with Periodic Sets. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 55–67. Springer, Heidelberg (1994)

[BW98] Boigelot, B., Wolper, P.: Verifying systems with infinite but regular state spaces. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 88–97. Springer, Heidelberg (1998)

[CC00] Comon, H., Cortier, V.: Flatness is not a weakness. In: Clote, P.G., Schwichtenberg, H. (eds.) CSL 2000. LNCS, vol. 1862, pp. 262–276. Springer, Heidelberg (2000)

[CGP00] Clarke, E., Grumberg, O., Peled, D.: Model checking. MIT Press (2000)

[CJ98] Comon, H., Jurski, Y.: Multiple counter automata, safety analysis and Presburger Arithmetic. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 268–279. Springer, Heidelberg (1998)

[CKL13] Carapelle, C., Kartzow, A., Lohrey, M.: Satisfiability of CTL* with constraints. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 455–469. Springer, Heidelberg (2013)

[Con12] Conchon, S.: SMT Techniques and their Applications: from Alt-Ergo to Cubicle. Habilitation à Diriger des Recherches, Université Paris-Sud (2012)

[Coo72] Cooper, D.: Theorem proving in arithmetic without multiplication. Machine Learning 7, 91–99 (1972)

[DD07] Demri, S., D'Souza, D.: An automata-theoretic approach to constraint LTL. I & C 205(3), 380–415 (2007)

[DDS12] Demri, S., Dhar, A.K., Sangnier, A.: Taming Past LTL and Flat Counter Systems. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 179–193. Springer, Heidelberg (2012)

[DDS13] Demri, S., Dhar, A.K., Sangnier, A.: On the complexity of verifying regular properties on flat counter systems, In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part II. LNCS, vol. 7966, pp. 162–173. Springer, Heidelberg (2013)

[DFGvD10] Demri, S., Finkel, A., Goranko, V., van Drimmelen, G.: Model-checking CTL^* over flat Presburger counter systems. JANCL 20(4), 313–344 (2010)

[DG08] Demri, S., Gascon, R.: Verification of qualitative Z constraints. TCS 409(1), 24–40 (2008)

[DG09] Demri, S., Gascon, R.: The effects of bounding syntactic resources on Presburger LTL. JLC 19(6), 1541–1575 (2009)

[DLN07] Demri, S., Lazić, R., Nowak, D.: On the freeze quantifier in constraint LTL: decidability and complexity. I & C 205(1), 2–24 (2007)

[DLS10] Demri, S., Lazić, R., Sangnier, A.: Model checking memoryful linear-time logics over one-counter automata. TCS 411(22-24), 2298–2316 (2010)

[dMB08] 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)

[Esp94] Esparza, J.: On the decidability of model checking for several μ-calculi and Petri nets. In: Tison, S. (ed.) CAAP 1994. LNCS, vol. 787, pp. 115–129. Springer, Heidelberg (1994)

[Fit02] Fitting, M.: Modal logic between propositional and first-order. JLC 12(6), 1017–1026 (2002)

[FL02] Finkel, A., Leroux, J.: How to compose Presburger-accelerations: Applications to broadcast protocols. In: Agrawal, M., Seth, A. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 145–156. Springer, Heidelberg (2002)

[FO97] Fribourg, L., Olsén, H.: Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 213–227. Springer, Heidelberg (1997)

[FR74] Fischer, M., Rabin, M.: Super-exponential complexity of Presburger arithmetic. In: Complexity of Computation. SIAM-AMS proceedings, vol. 7, pp. 27–42. AMS (1974)

[FR79] Ferrante, J., Rackoff, C.: The Computational Complexity of Logical Theories. Lecture Notes in Mathematics, vol. 718. Springer (1979)

[Fri00] Fribourg, L.: Petri nets, flat languages and linear arithmetic. In: 9th Workshop on Functional and Logic Programming (WFLP), pp. 344–365 (2000)

[FS01] Finkel, A., Schnoebelen, P.: Well-structured transitions systems everywhere! TCS 256(1-2), 63–92 (2001)

[FS08] Finkel, A., Sangnier, A.: Reversal-bounded counter machines revisited. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 323–334. Springer, Heidelberg (2008)

[FS09] Figueira, D., Segoufin, L.: Future-looking logics on data words and trees. In: Královič, R., Niwiński, D. (eds.) MFCS 2009. LNCS, vol. 5734, pp. 331–343. Springer, Heidelberg (2009)

[GI81] Gurari, E., Ibarra, O.: The complexity of decision problems for finite-turn multicounter machines. In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 495–505. Springer, Heidelberg (1981)

[GNRZ07] Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Combination methods for satisfiability and model-checking of infinite-state systems. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 362–378. Springer, Heidelberg (2007)

[Gor94] Goranko, V.: Temporal logic with reference pointers. In: Gabbay, D.M., Ohlbach, H.J. (eds.) ICTL 1994. LNCS (LNAI), vol. 827, pp. 133–148. Springer, Heidelberg (1994)

[Grä88] Grädel, E.: Subclasses of Presburger arithmetic and the polynomial-time hierarchy. TCS 56, 289–301 (1988)

[GS92] German, S., Sistla, P.: Reasoning about systems with many processes. JACM 39(3), 675–735 (1992)

[Hab97] Habermehl, P.: On the complexity of the linear-time mu-calculus for Petri nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 102–116. Springer, Heidelberg (1997)

[Hen90] Henzinger, T.: Half-order modal logic: how to prove real-time properties. In: PODC 1990, pp. 281–296. ACM Press (1990)

[HL11] Hague, M., Lin, A.W.: Model checking recursive programs with numeric data types. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 743–759. Springer, Heidelberg (2011)

[HP79] Hopcroft, J., Pansiot, J.J.: On the reachability problem for 5-dimensional vector addition systems. TCS 8, 135–159 (1979)

[HR87] Howell, R.R., Rosier, L.E.: An analysis of the nonemptiness problem for classes of reversal-bounded multicounter machines. JCSS 34(1), 55–74 (1987)

[HR89] Howell, R.R., Rosier, L.E.: Problems concerning fairness and temporal logic for conflict-free petri nets. TCS 64, 305–329 (1989)

[HRS76] Hunt, H., Rosenkrantz, D., Szymanski, T.: On the equivalence, containment, and covering problems for the regular and context-free languages. JCSS 12, 222–268 (1976)

[Iba78] Ibarra, O.: Reversal-bounded multicounter machines and their decision problems. JACM 25(1), 116–133 (1978)

[Jan90] Jančar, P.: Decidability of a temporal logic problem for Petri nets. TCS 74(1), 71–93 (1990)

[KM69] Karp, R.M., Miller, R.E.: Parallel program schemata. JCSS 3(2), 147–195 (1969)

[Kos82] Kosaraju, R.: Decidability of reachability in vector addition systems. In: STOC 1982, pp. 267–281 (1982)

[Ler03] Leroux, J.: Algorithmique de la vérification des systèmes à compteurs. Approximation et accélération. Implémentation de l'outil FAST. PhD thesis, ENS de Cachan, France (2003)

[Ler09] Leroux, J.: The general vector addition system reachability problem by Presburger inductive invariants. In: LICS 2009, pp. 4–13. IEEE (2009)

[Ler12] Leroux, J.: Presburger counter machines. Habilitation à Diriger des Recherches, Université Bordeaux (2012)

[Ler13] Leroux, J.: Presburger Vector Addition Systems. In: LICS 2013, pp. 23–32. IEEE (2013)

[Lip76] Lipton, R.J.: The reachability problem requires exponential space. Technical Report 62, Department of Computer Science, Yale University (1976)

[LP05] Lisitsa, A., Potapov, I.: Temporal logic with predicate λ-abstraction. In: TIME 2005, pp. 147–155. IEEE (2005)

[LP09] Leroux, J., Point, G.: TaPAS: The Talence Presburger Arithmetic Suite. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 182–185. Springer, Heidelberg (2009)

[LS05] Leroux, J., Sutre, G.: Flat counter automata almost everywhere! In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 489–503. Springer, Heidelberg (2005)

[Lut04] Lutz, C.: NEXPTIME-complete description logics with concrete domains. ACM ToCL 5(4), 669–705 (2004)

[May84] Mayr, E.: An algorithm for the general Petri net reachability problem. SIAM Journal of Computing 13(3), 441–460 (1984)

[May03] Mayr, R.: Undecidable problems in unreliable computations. TCS 297(1-3), 337–354 (2003)

[Min61] Minsky, M.: Recursive unsolvability of Post's problems of 'tag' and other topics in theory of Turing machines. Annals of Mathematics 74(3), 437–455 (1961)

[Min67] Minsky, M.: Computation, Finite and Infinite Machines. Prentice Hall (1967)

[Mon10] Monniaux, D.: Quantifier elimination by lazy model enumeration. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 585–599. Springer, Heidelberg (2010)

[MP92] Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer (1992)

[MP95] Manna, Z., Pnueli, A.: Temporal verification of reative systems: safety. Springer (1995)

[MS77] Mandel, A., Simon, I.: On finite semigroups of matrices. TCS 5(2), 101–111 (1977)

[Opp78] Oppen, D.: A $2^{2^{2pn}}$ upper bound on the complexity of Presburger arithmetic. JCSS 16(3), 323–332 (1978)

[OW05] Ouaknine, J., Worrell, J.: On the Decidability of Metric Temporal Logic. In: LICS 2005, pp. 188–197. IEEE (2005)

[Pnu77] Pnueli, A.: The temporal logic of programs. In: FOCS 1977, pp. 46–57. IEEE (1977)

[Pre29] Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes Rendus du Premier Congrès de Mathématiciens des Pays Slaves, Warszawa, pp. 92–101 (1929)

[Pug92] Pugh, W.: A practical algorithm for exact array dependence analysis. Communications of the ACM 35(8), 102–114 (1992)

[QR05] Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)

[Reu90] Reutenauer, C.: The mathematics of Petri nets. Masson and Prentice (1990)

[RL78] Reddy, C., Loveland, W.: Presburger arithmetic with bounded quantifier alternation. In: STOC 1978, pp. 320–325. ACM Press (1978)

[Sch02] Schnoebelen, P.: Verifying lossy channel systems has nonprimitive recursive complexity. IPL 83, 251–261 (2002)

[Sch10] Schnoebelen, P.: Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 616–628. Springer, Heidelberg (2010)

[Sho79] Shostak, R.: A practical decision procedure for arithmetic with function symbols. JACM 26(2), 351–360 (1979)

[SJ80] Suzuki, N., Jefferson, D.: Verification Decidability of Presburger Array Programs. JACM 27(1), 191–205 (1980)

[ST11] Segoufin, L., Torunczyk, S.: Automata based verification over linearly ordered data domains. In: STACS 2011, pp. 81–92 (2011)

Decidability and Complexity via Mosaics of the Temporal Logic of the Lexicographic Products of Unbounded Dense Linear Orders

Philippe Balbiani[1] and Szabolcs Mikulás[2]

[1] Institut de recherche en informatique de Toulouse, CNRS — University of Toulouse, 118 Route de Narbonne, 31062 Toulouse CEDEX 9, France
balbiani@irit.fr

[2] Department of Computer Science and Information Systems, Birbeck — University of London, Malet Street, London WC1E 7HX, UK
szabolcs@dcs.bbk.ac.uk

Abstract. This article considers the temporal logic of the lexicographic products of unbounded dense linear orders and provides via mosaics a complete decision procedure in nondeterministic polynomial time for the satisfiability problem it gives rise to.

Keywords: Linear temporal logic, lexicographic product, satisfiability problem, decidability, complexity, mosaic method, decision procedure.

1 Introduction

The mosaic method originates in algebraic logic, see [19], where the existence of a model is proved to be equivalent to the existence of a finite set of partial models verifying some conditions. It has also been applied for proving completeness and decidability of temporal logics over linear flows of time. See [5, section 6.4], or [7, 16, 17, 20]. For their use, specialized systems such as temporal logics must be combined with each other. This has led to the development of techniques for the combination of linear flows of time such as the classical operation of Cartesian product [10, 11, 14, 21]. Within the context of modal logic, the operation of lexicographic product of Kripke frames has been introduced as a variant of the operation of Cartesian product. It has also been used for defining the semantical basis of different languages designed for time representation and temporal reasoning from the perspective of non-standard analysis. See [1–3].

In [3], the temporal logic of the lexicographic products of unbounded dense linear orders has been considered and its complete axiomatization has been given. The purpose of this paper is to apply the mosaic method for providing a complete decision procedure in nondeterministic polynomial time for the satisfiability problem this temporal logic gives rise to. Its section-by-section breakdown is as follows. Section 2 formally introduces the lexicographic products of unbounded dense linear orders and presents the syntax and the semantics of the temporal logic we will be working with. Sections 3 defines mosaics and collections

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 151–164, 2013.

of mosaics satisfying saturation properties. In Section 4 and 5, we prove the completeness and soundness, respectively, of the mosaic method. Applying these result we prove in Section 6 that the satisfiability problem for our temporal logic is decidable in nondeterministic polynomial time.

2 Products of Unbounded Dense Linear Orders

Let $\mathcal{F}_1 = (T_1, <_1)$ and $\mathcal{F}_2 = (T_2, <_2)$ be linear orders. Their lexicographic product is the structure $\mathcal{F} = (T, \prec_1, \prec_2)$ where

- $T = T_1 \times T_2$,
- $\prec_1$ and $\prec_2$ are binary relations on T defined by $(s_1, s_2) \prec_1 (t_1, t_2)$ iff $s_1 <_1 t_1$ and $(s_1, s_2) \prec_2 (t_1, t_2)$ iff $s_1 = t_1$ and $s_2 <_2 t_2$.

We define the binary relation $\prec$ on T by $(s_1, s_2) \prec (t_1, t_2)$ iff $(s_1, s_2) \prec_1 (t_1, t_2)$ or $(s_1, s_2) \prec (t_1, t_2)$. The effect of the operation of lexicographic product may be described informally as follows: given two linear orders, their lexicographic product is the structure obtained by replacing each point of the first one by a copy of the second one. The global intuitions underlying such an operation is based upon the fact that, depending on the accuracy required or the available knowledge, one can describe a temporal situation at different levels of abstraction. See [4, section I.2.2], or [8] for details. In Fig. 1 below, we have $s_1 <_1 t_1$ and $s_2 <_2 t_2$. As a result, we have $(s_1, s_2) \prec_2 (s_1, t_2)$, $(s_1, s_2) \prec_1 (t_1, s_2)$, $(s_1, s_2) \prec_1 (t_1, t_2)$, $(s_1, t_2) \prec_1 (t_1, s_2)$, $(s_1, t_2) \prec_1 (t_1, t_2)$ and $(t_1, s_2) \prec_2 (t_1, t_2)$. It is now time to meet the temporal language we will be working with. Let At be a countable set of atomic formulas (with typical members denoted p, q, etc). We define the set $\mathcal{L}_t$ of formulas of our temporal language (with typical members denoted φ, ψ, etc.) as follows:

- $\varphi := p \mid \bot \mid \neg\varphi \mid (\varphi \vee \psi) \mid G_1\varphi \mid G_2\varphi \mid H_1\varphi \mid H_2\varphi$,

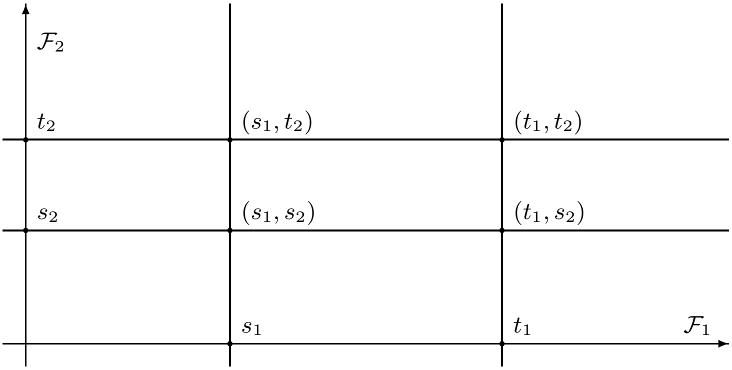


Fig. 1. Illustration of $\prec_1$ and $\prec_2$

the formulas $G_1\varphi$, $G_2\varphi$, $H_1\varphi$ and $H_2\varphi$ being read "φ will be true at each point within the future of but not infinitely close to the present point", "φ will be true at each instant within the future of and infinitely close to the present instant", "φ has been true at each point within the past of but not infinitely close to the present point" and "φ has been true at each point within the past of and infinitely close to the present point". We adopt the standard definitions for the remaining Boolean connectives. As usual, we define for all $i \in \{1,2\}$,

- $F_i\varphi := \neg G_i\neg\varphi$,
- $P_i\varphi := \neg H_i\neg\varphi$,
- $\Diamond_i\varphi := \varphi \vee F_i\varphi \vee P_i\varphi$.

The notion of a subformula is standard. It is usual to omit parentheses if this does not lead to any ambiguity. The size of a formula φ, in symbols $|\varphi|$, is the number of symbols of φ. A model is a structure $\mathcal{M} = (\mathcal{F}_1, \mathcal{F}_2, V)$ where $\mathcal{F}_1 = (T_1, <_1)$ and $\mathcal{F}_2 = (T_2, <_2)$ are linear orders and $V\colon At \to \wp(T_1 \times T_2)$ is a valuation. Satisfaction is a ternary relation $\models$ between a model $\mathcal{M} = (\mathcal{F}_1, \mathcal{F}_2, V)$, a pair $(s_1, s_2) \in T_1 \times T_2$ and a formula φ. It is defined by induction on φ as usual. In particular, for all $i \in \{1,2\}$,

- $\mathcal{M}, (s_1, s_2) \models G_i\varphi$ iff $\mathcal{M}, (t_1, t_2) \models \varphi$ for every $(t_1, t_2) \in T_1 \times T_2$ such that $(s_1, s_2) \prec_i (t_1, t_2)$,
- $\mathcal{M}, (s_1, s_2) \models H_i\varphi$ iff $\mathcal{M}, (t_1, t_2) \models \varphi$ for every $(t_1, t_2) \in T_1 \times T_2$ such that $(t_1, t_2) \prec_i (s_1, s_2)$.

As a result, for all $i \in \{1,2\}$,

- $\mathcal{M}, (s_1, s_2) \models F_i\varphi$ iff $\mathcal{M}, (t_1, t_2) \models \varphi$ for some $(t_1, t_2) \in T_1 \times T_2$ such that $(s_1, s_2) \prec_i (t_1, t_2)$,
- $\mathcal{M}, (s_1, s_2) \models P_i\varphi$ iff $\mathcal{M}, (t_1, t_2) \models \varphi$ for some $(t_1, t_2) \in T_1 \times T_2$ such that $(t_1, t_2) \prec_i (s_1, s_2)$,
- $\mathcal{M}, (s_1, s_2) \models \Diamond_2\varphi$ iff $\mathcal{M}, (s_1, t) \models \varphi$ for some $t \in T_2$.

$\mathcal{M}$ is said to be a model for φ iff there exists $(s_1, s_2) \in T_1 \times T_2$ such that $\mathcal{M}, (s_1, s_2) \models \varphi$. In this case, we shall also say that φ is satisfied in $\mathcal{M}$. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be classes of linear orders. We shall say that a formula φ is satisfiable with respect to $(\mathcal{C}_1, \mathcal{C}_2)$ iff there exists a linear order $\mathcal{F}_1 = (T_1, <_1)$ in $\mathcal{C}_1$, there exists a linear order $\mathcal{F}_2 = (T_2, <_2)$ in $\mathcal{C}_2$ and there exists a valuation $V\colon At \to \wp(T_1 \times T_2)$ such that $(\mathcal{F}_1, \mathcal{F}_2, V)$ is a model for φ. The temporal logic of $(\mathcal{C}_1, \mathcal{C}_2)$ is the set of all formulas φ such that $\neg\varphi$ is not satisfiable with respect to $(\mathcal{C}_1, \mathcal{C}_2)$. The class of all unbounded dense linear orders will be denoted $\mathcal{C}_{ud}$. [3] considers the temporal logic of $(\mathcal{C}_{ud}, \mathcal{C}_{ud})$ and gives its complete axiomatization. The satisfiability problem of this temporal logic is to

- determine whether a given formula φ is satisfiable with respect to $(\mathcal{C}_{ud}, \mathcal{C}_{ud})$.

In order to provide a complete decision procedure in nondeterministic polynomial time for it, we use mosaics.

3 Mosaics

Until the end of this paper, ξ will denote a fixed formula and Γ will denote the least set of formulas such that $\Diamond_2\xi \in \Gamma$ (recall that $\Diamond_2\varphi$ is defined as $\varphi \vee F_2\varphi \vee P_2\varphi$) and $\top \in \Gamma$, Γ is closed under subformulas and single negations (we identify $\neg\neg\gamma$ with γ) and

- if $G_1\varphi \in \Gamma$ or $H_1\varphi \in \Gamma$, then $G_2\varphi, H_2\varphi \in \Gamma$,
- if $F_1\varphi \in \Gamma$ or $P_1\varphi \in \Gamma$, then $\Diamond_2\varphi \in \Gamma$.

Recall that ξ has at most $|\xi|$ subformulas. Closing this set under single negations gives us at most $2 \times |\xi|$ formulas. Then $\Diamond_2\xi$ yields $2 \times |\xi| + 10$ formulas (subformulas and their negations). The first requirement above introduces at most 2 new formulas plus their negations, and the last requirement introduces at most 10 new formulas (with negations), for every φ. Thus we get that $|\Gamma| \leq 14 \times (2 \times |\xi| + 10) + 2$ (the 2 is for $\top$ and $\bot$).

Let λ be a function such that $\mathrm{dom}(\lambda) \subseteq \Gamma$ is closed under single negations and $\mathrm{ran}(\lambda) \subseteq \{0, 1\}$. We say that λ is *adequate* if

- $\top \in \mathrm{dom}(\lambda)$ and $\lambda(\top) = 1$,
- for every $\gamma \in \mathrm{dom}(\lambda)$, we have $\lambda(\neg\gamma) = 1 - \lambda(\gamma)$,
- for every $\gamma \vee \rho \in \mathrm{dom}(\lambda)$, we have $\lambda(\gamma \vee \rho) \geq \lambda(\gamma)$ provided that $\gamma \in \mathrm{dom}(\lambda)$, $\lambda(\gamma \vee \rho) \geq \lambda(\rho)$ provided that $\rho \in \mathrm{dom}(\lambda)$, and $\lambda(\gamma \vee \rho) = \max\{\lambda(\gamma), \lambda(\rho)\}$ if both $\gamma, \rho \in \mathrm{dom}(\lambda)$.

The reason for the complicated form of the last condition is that generally we do not require that $\mathrm{dom}(\lambda)$ is closed under subformulas. However, when we state a requirement that $\lambda(\gamma) \in \{0, 1\}$, then we implicitly require that $\gamma \in \mathrm{dom}(\lambda)$.

Definition 1. *Let $i \in \{1, 2\}$ and (σ, τ) be a pair of adequate functions. We define the following* coherence properties.

G_i-coherence $\sigma(G_i\varphi) = 1$ *implies* $\tau(G_i\varphi) = \tau(\varphi) = 1$.
H_i-coherence $\tau(H_i\varphi) = 1$ *implies* $\sigma(H_i\varphi) = \sigma(\varphi) = 1$.

1. *A* 1-mosaic *is a pair (σ, τ) such that σ and τ are adequate functions, (σ, τ) satisfies G_1- and H_1-coherence and the following* transfer *conditions.*
   G-transfer $\sigma(G_1\varphi) = 1$ *implies* $\tau(G_2\varphi) = \tau(H_2\varphi) = 1$.
   H-transfer $\tau(H_1\varphi) = 1$ *implies* $\sigma(G_2\varphi) = \sigma(H_2\varphi) = 1$.
2. *A* 2-mosaic *is a pair (σ, τ) such that σ and τ are adequate functions with full domain* $\mathrm{dom}(\sigma) = \mathrm{dom}(\tau) = \Gamma$*, (σ, τ) satisfies G_2- and H_2-coherence and the following* uniformity *conditions.*
   G_1-uniformity $\sigma(G_1\varphi) = \tau(G_1\varphi)$.
   H_1-uniformity $\sigma(H_1\varphi) = \tau(H_1\varphi)$.

As an example of mosaics take a model $\mathcal{M} = (\mathcal{F}_1, \mathcal{F}_2, V)$ with unbounded, dense linear orders $\mathcal{F}_1 = (T_1, <_1)$ and $\mathcal{F}_2 = (T_2, <_2)$ and a valuation $V\colon At \to$

$\wp(T_1 \times T_2)$. Denote $T := T_1 \times T_2$. Define for every $(s,t) \in T$, $\lambda_{(s,t)} \colon \mathit{\Gamma} \to \{0,1\}$ by

$$\lambda_{(s,t)}(\gamma) := \begin{cases} 1 & \text{if } \mathcal{M},(s,t) \models \gamma \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

for every $\gamma \in \mathit{\Gamma}$. It is straightforward to check that

- for every $(s,t),(u,v) \in T$ with $s <_1 u$, the pair $(\lambda_{(s,t)}, \lambda_{(u,v)})$ is a 1-mosaic (with full domain)
- for every $(s,t),(s,u) \in T$ with $t <_2 u$, the pair $(\lambda_{(s,t)}, \lambda_{(s,u)})$ is a 2-mosaic.

For every $s \in T_1$, define κ_s as follows:

$$\kappa_s(\gamma) := \begin{cases} 1 & \text{if } \lambda_{(s,u)}(\gamma) = 1 \text{ for every } u \in T_2 \\ 0 & \text{if } \lambda_{(s,u)}(\gamma) = 0 \text{ for every } u \in T_2 \\ \text{undefined} & \text{otherwise} \end{cases} \tag{2}$$

for every $\gamma \in \mathit{\Gamma}$. Then κ_s is an adequate function and (κ_s, κ_t) is a 1-mosaic whenever $s <_1 t$. An i-SSM will correspond to a flow of time in dimension i.

Definition 2. *Let $i \in \{1,2\}$. An* i-saturated set of mosaics*, an i-SSM, is a collection M of i-mosaics such that M satisfies the Density, No-endpoints and the corresponding* i-saturation *conditions below.*

Density *If $(\sigma,\tau) \in M$, then there is μ such that $(\sigma,\mu),(\mu,\tau) \in M$.*

No-endpoints *If $(\sigma,\tau) \in M$, then there are μ,ν such that $(\mu,\sigma) \in M$ and $(\tau,\nu) \in M$.*

F_1-saturation—insertion *If $(\sigma,\tau) \in M$, $\sigma(F_1\varphi) = 1$ and $\tau(F_1\varphi) = \tau(\Diamond_2\varphi) = 0$, then there is μ such that $\mu(\Diamond_2\varphi) = 1$ and $(\sigma,\mu),(\mu,\tau) \in M$.*

F_2-saturation—insertion *If $(\sigma,\tau) \in M$, $\sigma(F_2\varphi) = 1$ and $\tau(F_2\varphi) = \tau(\varphi) = 0$, then there is μ such that $\mu(\varphi) = 1$ and $(\sigma,\mu),(\mu,\tau) \in M$.*

P_1-saturation—insertion *If $(\sigma,\tau) \in M$, $\tau(P_1\varphi) = 1$ and $\sigma(P_1\varphi) = \sigma(\Diamond_2\varphi) = 0$, then there is μ such that $\mu(\Diamond_2\varphi) = 1$ and $(\sigma,\mu),(\mu,\tau) \in M$.*

P_2-saturation—insertion *If $(\sigma,\tau) \in M$, $\tau(P_2\varphi) = 1$ and $\sigma(P_2\varphi) = \sigma(\varphi) = 0$, then there is μ such that $\mu(\varphi) = 1$ and $(\sigma,\mu),(\mu,\tau) \in M$.*

F_1-saturation—expansion *If $(\sigma,\tau) \in M$ and $\tau(F_1\varphi) = 1$, then there is μ such that $\mu(\Diamond_2\varphi) = 1$ and $(\tau,\mu) \in M$.*

F_2-saturation—expansion *If $(\sigma,\tau) \in M$ and $\tau(F_2\varphi) = 1$, then there is μ such that $\mu(\varphi) = 1$ and $(\tau,\mu) \in M$.*

P_1-saturation—expansion *If $(\sigma,\tau) \in M$ and $\sigma(P_1\varphi) = 1$, then there is μ such that $\mu(\Diamond_2\varphi) = 1$ and $(\mu,\sigma) \in M$.*

P_2-saturation—expansion *If $(\sigma,\tau) \in M$ and $\sigma(P_2\varphi) = 1$, then there is μ such that $\mu(\varphi) = 1$ and $(\mu,\sigma) \in M$.*

That is, besides the Density and No-endpoint conditions, a 1-SSM should satisfy the F_1-saturation and P_1-saturation conditions and a 2-SSM should satisfy the F_2-saturation and P_2-saturation conditions. We say that M is an i-SSM for φ if there is $(\mu, \nu) \in M$ such that $\mu(\varphi) = 1$ or $\nu(\varphi) = 1$. Let us continue with our example. The collections

$$\{(\lambda_{(s,t)}, \lambda_{(u,v)}) : (s,t) \prec_1 (u,v)\} \text{ and } \{(\kappa_s, \kappa_u) : s <_1 u\}$$

of 1-mosaics are 1-SSMs. Fix $s \in T_1$ and consider the set M_s of 2-mosaics defined by

$$M_s := \{(\lambda_{(s,t)}, \lambda_{(s,u)}) : t, u \in T_2, t <_2 u\}.$$

It is easy to check that M_s is a 2-SSM for every $s \in T_1$. A 1-supermosaic will be a 1-mosaic of two 2-SSMs.

Definition 3. *Let M be a 2-SSM. We define λ_M by*

$$\lambda_M(\gamma) := \begin{cases} 1 & \textit{if } \mu(\gamma) = \nu(\gamma) = 1 \textit{ for every } (\mu,\nu) \in M \\ 0 & \textit{if } \mu(\gamma) = \nu(\gamma) = 0 \textit{ for every } (\mu,\nu) \in M \\ \textit{undefined} & \textit{otherwise} \end{cases} \tag{3}$$

for every $\gamma \in \Gamma$. Observe that λ_M is an adequate function.

A 1-supermosaic *is a pair (M, N) of 2-SSMs such that (λ_M, λ_N) is a 1-mosaic.*

In our example, for every $s, t \in T_1$ such that $s <_1 t$, the pair (M_s, M_t) is a 1-supermosaic. A saturated set of 1-supermosaics will correspond to a flow (in dimension 1) of flows (in dimension 2).

Definition 4. *A* saturated set of 1-supermosaics*, a 1-SSS, is a collection Σ of 1-supermosaics such that $\{(\lambda_M, \lambda_N) : (M,N) \in \Sigma\}$ is a 1-SSM.*

We say that Σ is a 1-SSS for φ if there is $(M, N) \in \Sigma$ such that $\lambda_M(\Diamond_2\varphi) = 1$ or $\lambda_N(\Diamond_2\varphi) = 1$. Observe that then there is a 2-mosaic (σ, τ) in one of the 2-SSMs in one of the 1-supermosaics of Σ such that $\sigma(\varphi) = 1$ or $\tau(\varphi) = 1$. In our example the set

$$\Sigma_{\mathcal{M}} := \{(M_s, M_t) : s, t \in T_1, s <_1 t\}$$

is a 1-SSS and in fact a 1-SSS for ξ if there is $(s,t) \in T_1 \times T_2$ such that $\mathcal{M}, (s,t) \models \xi$. Our running example should convince the reader that satisfiability of ξ implies the existence of a 1-SSS for ξ. But we want to be more economical in creating the 1-SSS; we will describe the procedure for creating a smaller 1-SSS in Section 5. First we show how to create a model from a 1-SSS, though.

4 Completeness

In this section we show the completeness of the mosaic approach. We will need the following.

Definition 5. *Let $i \in \{1, 2\}$ and $\mathcal{W} = (W, <, \lambda)$ be a structure such that $(W, <)$ is a linear order and λ_q is an adequate function for every $q \in W$. We say that $\mathcal{W}$ is i-*consistent *if it satisfies for every $q \in W$, the corresponding i-*completness *and i-*soundness *conditions below.*

G_i-soundness *If $\lambda_q(G_i\gamma) = 1$, then $\lambda_p(G_i\gamma) = \lambda_p(\gamma) = 1$ for every $p \in W$ such that $q < p$.*

H_i-soundness *If $\lambda_q(H_i\rho) = 1$, then $\lambda_p(H_i\rho) = \lambda_p(\rho) = 1$ for every $p \in W$ such that $p < q$.*

F_1-completeness *If $\lambda_q(F_1\gamma) = 1$, then there is $p \in W$ such that $q < p$ and $\lambda_p(\Diamond_2\gamma) = 1$.*

F_2-completeness *If $\lambda_q(F_2\gamma) = 1$, then there is $p \in W$ such that $q < p$ and $\lambda_p(\gamma) = 1$.*

P_1-completeness *If $\lambda_q(P_1\rho) = 1$, then there is $p \in W$ such that $p < q$ and $\lambda_p(\Diamond_2\gamma) = 1$.*

P_2-completeness *If $\lambda_q(P_2\rho) = 1$, then there is $p \in W$ such that $p < q$ and $\lambda_p(\gamma) = 1$.*

That is, a 1-consistent structure must satisfy the G_1-soundness, H_1-soundness, F_1-completeness and P_1-completeness conditions, and a 2-consistent structure satisfies the G_2-soundness, H_2-soundness, F_2-completeness and P_2-completeness conditions.

Let $\mathcal{W} = (W, <, \lambda)$ be a structure such that $(W, <)$ is a linear order and λ_q is an adequate function for every $q \in W$. A *future defect* of $\mathcal{W}$ is a pair (q, γ) such that $\lambda_q(F_i\gamma) = 1$ but there is no $p > q$ such that λ_p satisfies the requirements in the F_i-completeness condition above. Past defects are defined similarly. Below we will construct a i-complete (i.e., without defects) and i-sound structure from an i-SSM, see Case 1 and 2 in the proof of Lemma 1 below, where we construct the required future and past witnesses for the defects. In addition, we need that the constructed structure is dense and without endpoints. That is why we will need Case 3 and 4, where we construct new successors and predecessors for each point in the linear order, which in the limit of the construction yields a dense linear order without endpoints.

Lemma 1. *Let $i \in \{1, 2\}$. Assume that M is an i-SSM for φ. Then there is an i-consistent structure $\mathcal{Q}_M = (Q_M, <, \lambda)$ such that $(Q_M, <)$ is isomorphic to the rationals $\mathbb{Q}$ and $\lambda_q(\varphi) = 1$ for some $q \in Q_M$.*

Proof. We will define the order $(Q_M, <)$ and the adequate functions λ_q by induction. To this end let us have a countable enumeration D of potential defects $\{(q, \gamma, k) : q \in \mathbb{Q}, \gamma \in \varGamma, k \in \{1, 2, 3, 4\}\}$ such that every item appears infinitely often. The value of k will indicate the type of the potential defect: future, past, successor, predecessor.

By assumption there is an i-mosaic $(\mu, \nu) \in M$ such that $\mu(\varphi) = 1$ or $\nu(\varphi) = 1$. In the base step of the construction we define the finite order $Q_1 = \{0, 1\}$ with $0 < 1$ and functions $\lambda_0 = \mu$ and $\lambda_1 = \nu$. Obviously the soundness conditions restricted to Q_1 hold.

For the inductive step assume that we constructed a sound structure $\mathcal{Q}_n$ consisting of a finite order $(Q_n, <) = (q_0 < q_1 < \ldots < q_n)$ and adequate functions λ_q for $q \in Q_n$ such that $(\lambda_{q_j}, \lambda_{q_{j+1}}) \in M$ for every $j < n$. Let $D(n) = (q, \gamma, k)$. If $q \notin Q_n$, then we define $\mathcal{Q}_{n+1} := \mathcal{Q}_n$. Otherwise we consider the following four cases. If none of the four cases below holds, then we let $\mathcal{Q}_{n+1} := \mathcal{Q}_n$.

Case 1 $k = 1$, $\lambda_q(F_i\gamma) = 1$ and for every $r \in Q_n$ with $q < r$, we have $\lambda_r(\Diamond_2\gamma) = 0$ in case $i = 1$, or $\lambda_r(\gamma) = 0$ in case $i = 2$.

We will construct the required witness in the future of q. First assume that for every r with $q < r$, we have $\lambda_r(F_i\gamma) = 1$ (note that this includes the case $q = q_n$). The i-mosaic $(\lambda_{q_{n-1}}, \lambda_{q_n}) \in M$. By F_i-saturation—expansion there is an i-mosaic $(\lambda_{q_n}, \mu) \in M$ such that

- $\mu(\Diamond_2\gamma) = 1$ if $i = 1$
- $\mu(\gamma) = 1$ if $i = 2$.

In this case we define $(Q_{n+1}, <) := (q_0 < q_1 < \ldots < q_n < p)$ for some $p \in \mathbb{Q}$ such that $q_n < p$ and let $\lambda_p := \mu$. Next assume that there is r with $q < r$ such that $\lambda_r(F_i\gamma) = 0$. Let m be such that q_m is minimal in Q_n with respect to this property. Consider the i-mosaic $(\lambda_{q_{m-1}}, \lambda_{q_m}) \in M$. By F_i-saturation—insertion there is an adequate function μ such that $(\lambda_{q_{n-1}}, \mu), (\mu, \lambda_{q_n}) \in M$ and

- $\mu(\Diamond_2\gamma) = 1$ if $i = 1$
- $\mu(\gamma) = 1$ if $i = 2$.

Then we let $(Q_{n+1}, <) := (q_0 < \ldots < q_{m-1} < p < q_m < \ldots < q_n)$ for some $p \in \mathbb{Q}$ such that $q_{m-1} < p < q_m$ and define $\lambda_p := \mu$.

Case 2 $k = 2$, $\lambda_q(P_i\gamma) = 1$ and for every $r \in Q_n$ with $r < q$ we have $\lambda_r(\Diamond_2\gamma) = 0$ if $i = 1$, or $\lambda_r(\gamma) = 0$ if $i = 2$.

A completely analogous construction to Case 1 provides the required witness in the past of q.

Case 3 $k = 3$.

We will construct a new successor for q. In the case $q = q_n$, consider $(\lambda_{q_{n-1}}, \lambda_{q_n}) \in M$. By the No-endpoints condition, we have an i-mosaic $(\lambda_{q_n}, \mu) \in M$. We define $(Q_{n+1}, <) := (q_0 < q_1 < \ldots < q_n < p)$ for some $p \in \mathbb{Q}$ such that $q_n < p$ and let $\lambda_p := \mu$. Now assume that $q = q_m < q_n$. Consider the i-mosaic $(\lambda_{q_m}, \lambda_{q_{m+1}}) \in M$. Then there is an adequate function μ such that $(\lambda_{q_m}, \mu), (\mu, \lambda_{q_{m+1}}) \in M$, by the Density condition. Then we let $(Q_{n+1}, <) := (q_0 < \ldots < q_m < p < q_{m+1} < \ldots < q_n)$ for some $p \in \mathbb{Q}$ such that $q_{m-1} < p < q_m$ and define $\lambda_p := \mu$.

Case 4 $k = 4$.

A completely analogous construction to Case 3 provides a new predecessor of q.

It is easy to check that $\mathcal{Q}_{n+1}$ is sound in every case, since it consists of elements of M. Let $\mathcal{Q}_M = (Q_M, <) := \bigcup_{n \in \omega} \mathcal{Q}_n$ (recall that we defined λ_q in the step we created q for every $q \in Q_M$). It is easy to see that $(Q_M, <)$ is a countable linear order which is dense and does not have endpoints (by Case 3 and 4), hence isomorphic to $\mathbb{Q}$. Since we considered every potential defects infinitely often, it follows that $\mathcal{Q}_M$ does not contain any future or past defect in dimension i. That is, if $i = 1$ and $\lambda_q(F_1\gamma) = 1$, then there is $p \in Q_M$ such that $q < p$ and $\lambda_p(\Diamond_2\gamma) = 1$, and if $i = 2$ and $\lambda_q(F_2\gamma) = 1$, then there is $p \in Q_M$ such that $q < p$ and $\lambda_p(\gamma) = 1$ (and similarly for past formulas). Hence $\mathcal{Q}_M$ is i-consistent. □

Next we apply Lemma 1 in both dimensions to construct a model from a 1-SSS.

Lemma 2. *If there is a 1-SSS for ξ, then there is a model $\mathcal{M}$ for ξ. Furthermore, $\mathcal{M}$ can be chosen to be the lexicographic product of the rationals with some valuation V: $\mathcal{M} = (\mathbb{Q}, \mathbb{Q}, V)$.*

Proof. Let Σ be a 1-SSS for ξ. Thus Σ is a collection of 1-supermosaics (M, N) such that $\{(\lambda_M, \lambda_N) : (M, N) \in \Sigma\}$ is a 1-SSM. Furthermore, there is $(M, N) \in \Sigma$ such that $\lambda_M(\Diamond_2 \xi) = 1$ or $\lambda_N(\Diamond_2 \xi) = 1$.

We apply Lemma 1 to get the 1-consistent structure $\mathcal{Q}_\Sigma = (Q_\Sigma, <_1, \lambda)$ such that $(Q_\Sigma, <_1)$ is isomorphic to $\mathbb{Q}$ and $\lambda_q(\Diamond_2 \xi) = 1$ for some $q \in Q_\Sigma$. By the construction of $\mathcal{Q}_\Sigma$, for every $q \in Q_\Sigma$, there is a 2-SSM M such that $\lambda_q = \lambda_M$. Thus we can assume that there is a function $f \colon q \mapsto M$ with domain Q_Σ. For every $q \in Q_\Sigma$, we apply Lemma 1 to $f(q) = M$. Hence, we get a 2-consistent structure $\mathcal{Q}_{f(q)} = (Q_{f(q)}, <_2, \lambda)$ such that $(Q_{f(q)}, <_2)$ is isomorphic to $\mathbb{Q}$, and for every $r_q \in Q_{f(q)}$, the adequate function λ_{r_q} has full domain Γ (since $f(q) = M$ is a 2-SSM).

Let us replace every $q \in Q_\Sigma$ with the copy $(Q_{f(q)}, <_2)$ of $\mathbb{Q}$ (say, mapping q to 0). Thus we get a grid $\mathbb{Q} \times \mathbb{Q}$ such that the elements (q, r) have the property that $r = r_q \in Q_{f(q)}$. Hence to every $(q, r) \in \mathbb{Q} \times \mathbb{Q}$ we can associate a full adequate function $\lambda_{(q,r)} := \lambda_{r_q}$.

Let $\mathcal{M} = (\mathbb{Q}, \mathbb{Q}, V)$ be the model defined by the valuation V:

$$V(p) := \{(q, r) \in \mathbb{Q} \times \mathbb{Q} : \lambda_{(q,r)}(p) = 1\}$$

for every atomic proposition p. An easy formula-induction, using 1-consistency of $(Q_\Sigma, <_1, \lambda)$ and 2-consistency of $(Q_{f(q)}, <_2)$, establishes that

$$\mathcal{M}, (q, r) \models \varphi \text{ iff } \lambda_{(q,r)}(\varphi) = 1$$

for every $\varphi \in \Gamma$. Since we have $\lambda_q(\Diamond_2 \xi) = 1$ for some $q \in Q_\Sigma$, we also get that $\lambda(q, r)(\xi) = 1$ for some $r \in Q_{f(q)}$ by F_2/P_2-completeness. Hence $\mathcal{M}, (q, r) \models \xi$, that is, $\mathcal{M}$ is a model satisfying ξ. □

5 Soundness

For the reverse direction we also compute an upper bound on the size of the required 1-SSS.

Definition 6. *Let $\mathcal{W}_i = (W_i, <_i, \lambda)$ be an i-consistent structure. For $i = 1$ we define the following* transfer *conditions: for every $p, q \in W_1$ such that $p <_1 q$,*

G-transfer $\lambda_p(G_1 \varphi) = 1$ *implies* $\lambda_q(G_2 \varphi) = \lambda_q(H_2 \varphi) = 1$,
H-transfer $\lambda_q(H_1 \varphi) = 1$ *implies* $\lambda_p(G_2 \varphi) = \lambda_p(H_2 \varphi) = 1$.

For $i = 2$ we define the following uniformity *conditions: for every $p, q \in W_2$,*

G_1-uniformity $\lambda_p(G_1 \varphi) = \lambda_q(G_1 \varphi)$,
H_1-uniformity $\lambda_p(H_1 \varphi) = \lambda_q(H_1 \varphi)$.

We will need the following technical lemma.

Lemma 3. *Fix $i \in \{1, 2\}$. Let $\mathcal{W}_i = (W_i, <_i, \lambda)$ be an i-consistent structure such that $(W_i, <_i)$ is a dense, linear order without endpoints. Assume that $\mathcal{W}_i$ satisfies the G- and H-transfer conditions if $i = 1$, and the G_1- and H_1-uniformity conditions if $i = 2$.*

Let $u \in W_i$ and $\gamma \in \Gamma$ such that $\lambda_u(\gamma) = 1$. Then there is an i-SSM M_i for γ of size at most $(4 \times |\Gamma|)^2$. In fact, M_i can be chosen such that for some $U_i \subseteq W_i$ with $|U_i| \leq 4 \times |\Gamma|$

$$M_i = \{(\lambda_u, \lambda_v) : u, v \in U_i, (\exists u' \in W_i)(\exists v' \in W_i) u \equiv u' \ \&\ v \equiv v' \ \&\ u' <_i v'\}$$

where $w \equiv w'$ iff $\lambda_w = \lambda_{w'}$.

Proof. For every $w, w' \in W_i$, we let $w \equiv w'$ iff $\lambda_w = \lambda_{w'}$. Note that there are finitely many equivalence classes, since Γ is finite. For every formula $\varphi \in \Gamma$, let $W_i(\varphi) := \{w \in W_i : \lambda_w(\varphi) = 1\}$. Let $\overline{w}_\varphi$ be a *maximal* element of $W_i(\varphi)$ (provided that $W_i(\varphi)$ is not empty) in the following sense:

$$(\forall w' \in W_i(\varphi))(\exists w'' \in W_i(\varphi)) w' \leq_i w'' \ \&\ w'' \equiv \overline{w}_\varphi.$$

The existence of a maximal element can be easily shown. For every $\varphi \in \Gamma$ choose a maximal element $\overline{w}_\varphi$ from W_φ. Similarly, for every $\varphi \in \Gamma$, choose a minimal element $\underline{w}_\varphi$ from $W_i(\varphi)$. Let

$$W_i^- = \{\overline{w}_\varphi, \underline{w}_\varphi : \varphi \in \Gamma\}.$$

Note that $|W_i^-| \leq 2 \times |\Gamma|$.

The problem with W_i^- is that it may not contain "enough" points to cure density defects. Indeed, consider the *unique points* in W_i^-, i.e., those $w \in W_i^-$ such that for every $w' \neq w$, $\lambda_w \neq \lambda_{w'}$. Note that the set X of unique points can be linearly ordered: $x_1 <_i x_2 <_i \ldots <_i x_m$. Now consider two unique points $x_j, x_{j+1} \in W_i^-$ such that there is no $z \in W_i$ with $x_j <_i z <_i x_{j+1}$ in W_i and $z \equiv z' \in W_i^-$ for some z'. Then we would not be able to insert a point into the mosaic $(\lambda_{x_j}, \lambda_{x_{j+1}})$.

So let us expand W_i^- with the required witnesses for density defects. Take the enumeration $x_0 <_i x_1 <_i \ldots <_i x_m$ of unique points. Since $<_i$ is a dense order, there are infinitely many points in each open interval $]x_j, x_{j+1}[= \{x : x_j <_i x <_i x_{j+1}\}$. Thus, we can choose a point $s \in]x_j, x_{j+1}[$ such that there are infinitely many points t in $]x_j, x_{j+1}[$ with $\lambda_s = \lambda_t$. Let us denote such a chosen s by s_j for every $0 \leq j < m$. Define $U_i := W_i^- \cup \{s_j : 0 \leq j < m\}$. Note that $|U_i| \leq 4 \times |\Gamma|$. We claim that

$$M_i := \{(\lambda_u, \lambda_v) : u, v \in U_i, (\exists u' \in W_i)(\exists v' \in W_i) u \equiv u' \ \&\ v \equiv v' \ \&\ u' <_i v'\}$$

is the required i-SSM. Clearly, $|M_i| \leq (4 \times |\Gamma|)^2$. The elements of M_i are G_i- and H_i-coherent because $\mathcal{W}_i$ is sound. The transfer (for $i = 1$) and uniformity (for $i = 2$) conditions also hold, since $\mathcal{W}_i$ has the corresponding properties. Thus every element of M_i is an i-mosaic. It remains to show the saturation conditions.

For the F_i-saturation—expansion requirement assume that $(\lambda_u, \lambda_v) \in M$ and $\lambda_v(F_i\varphi) = 1$. Let $v' \in W$ such that $v \equiv v'$. Since $\lambda_{v'}(F_i\varphi) = 1$ and $\mathcal{W}_i$ is F_i-complete, there is $z \in W_i$ such that $v' <_i z$ and

- $\lambda_z(\Diamond_2\varphi) = 1$ in case $i = 1$,
- $\lambda_z(\varphi) = 1$ in case $i = 2$.

Let $\overline{w}$ be the maximal element in $W_1(\Diamond_2\varphi)$ or $W_2(\varphi)$ (depending on the value of i) that we put in U_i. By the maximality of $\overline{w}$, there is $z' \in W_i$ such that $z \leq_i z'$ and $z' \equiv \overline{w}$. By this observation, we get that $(\lambda_v, \lambda_{\overline{w}}) \in M_i$ as required.

For the F_i-saturation—insertion requirement we work out only the case $i = 2$, since the case $i = 1$ is completely analogous. So assume that $(\lambda_u, \lambda_v) \in M_2$, $\lambda_u(F_2\varphi) = 1$ and $\lambda_v(F_2\varphi) = \lambda_v(\varphi) = 0$. Let $u', v' \in W_2$ such that $u \equiv u'$, $v \equiv v'$ and $u' <_2 v'$. Since $\mathcal{W}_2$ is F_2-complete, there is $z \in W_2$ such that $u' <_2 z$ and $\lambda_z(\varphi) = 1$. Let $\overline{w}$ be the maximal element of $W_2(\varphi)$ that we put in U_2. Then there is $z' \in W_2$ such that $z \leq_2 z'$ and $z' \equiv \overline{w}$. Hence $(\lambda_u, \lambda_{\overline{w}}) \in M_2$. Furthermore, $z' <_2 v'$, since $\lambda_{v'}(F_2\varphi) = \lambda_{v'}(\varphi) = 0$ and $\mathcal{W}_2$ is G_2-sound. That is, $(\lambda_{\overline{w}}, \lambda_v) \in M_2$ as well. Thus we can insert the appropriate mosaics into (λ_u, λ_v).

Checking the saturation conditions for past formulas is completely analogous. The No-endpoints requirement follows from the fact that $(W_i, <_i)$ is an unbounded linear order. Indeed, let $(\lambda_u, \lambda_v) \in M_i$ and $v' \in W_i$ such that $v \equiv v'$. Then there is $w \in W_i$ such that $v' <_i w$ and $\lambda_w(\top) = 1$. Let $\overline{w}$ be the be the maximal element of $W_i(\top)$. Then $(\lambda_v, \lambda_{\overline{w}}) \in M_i$ as required.

It remains to show the Density condition. Let $(\lambda_u, \lambda_v) \in M_i$ be an arbitrary mosaic and $u', v' \in W_i$ such that $u' <_i v'$, $u \equiv u'$ and $v \equiv v'$. If either u' or v' is not unique, then we can insert either (λ_u, λ_u) or (λ_v, λ_v) into (λ_u, λ_v). So assume that both u' and v' are unique, say $u' = x_j$ and $v' = x_{j+k}$. But we defined $s_j \in]x_j, x_{j+1}[$ in this case, and we have $(\lambda_u, \lambda_{s_j})$ and $(\lambda_{s_j}, \lambda_v)$ in M_i. Hence we can insert the required mosaics into (λ_u, λ_v) in this case as well.

Finally note that we chose a representative from the equivalence class $W_i(\gamma)$, hence M_i is indeed an i-SSM for γ. □

We are ready to state the reverse of Lemma 2. In the proof, we will apply Lemma 3 in both the "vertical" and "horizontal" dimensions.

Lemma 4. *If ξ is satisfiable, then there is a 1-SSS Σ_1 for ξ of size polynomial in terms of the size of ξ. In fact, the number of elements in Σ_1 is bounded by $(4 \times |\Gamma|)^4$.*

Proof. Assume that ξ is satisfied in a model, say, $\mathcal{M} = ((T_1, <_1), (T_2, <_2), V)$ and $\mathcal{M},(s,t) \models \xi$. We recall the definition of $\lambda_{(p,q)}$ from (1): for every $(p, q) \in T_1 \times T_2$,

$$\lambda_{(p,q)}(\gamma) = \begin{cases} 1 & \text{if } \mathcal{M}, (p,q) \models \gamma \\ 0 & \text{otherwise} \end{cases}$$

for every $\gamma \in \Gamma$. In particular, $\lambda_{(s,t)}(\xi) = 1$.

For every $p \in T_1$, consider $\mathcal{W}_2^p = (\{p\} \times T_2, \prec_2, \lambda)$ (where $(p,q) \prec_2 (p,r)$ iff $q <_2 r$). Observe that $(p,q) \models G_1\varphi$ iff $(p,r) \models G_1\varphi$ (and $(p,q) \models H_1\varphi$ iff $(p,r) \models H_1\varphi$) for every $q, r \in T_2$. Hence $\mathcal{W}_2^p$ is a 2-consistent structure that satisfies G_1- and H_1-uniformity. By applying Lemma 3 we get a 2-SSM M_2^p for ξ (and in fact for $\Diamond_2\xi$) such that

$$M_2^p = \{(\lambda_{(p,q)}, \lambda_{(p,r)}) : q, r \in U_2^p, (\exists q' \in T_2)(\exists r' \in T_2) q \equiv q' \ \& \ r \equiv r' \ \& \ q' <_2 r'\}$$

where $u \equiv v$ iff $\lambda_{(p,u)} = \lambda_{(p,v)}$, and $U_2^p \subseteq T_2$ such that $|U_2^p| \leq 4 \times |\Gamma|$.

Next we recall the definition λ_M for 2-mosaics M from (3):

$$\lambda_M(\gamma) := \begin{cases} 1 & \text{if } \mu(\gamma) = \nu(\gamma) = 1 \text{ for every } (\mu, \nu) \in M \\ 0 & \text{if } \mu(\gamma) = \nu(\gamma) = 0 \text{ for every } (\mu, \nu) \in M \\ \text{undefined} & \text{otherwise} \end{cases}$$

for every $\gamma \in \Gamma$. Note that $\lambda_N(\Diamond_2\xi) = 1$ for $N = M_2^s$. We define $\lambda_p := \lambda_N$ with $N = M_2^p$ for every $p \in T_1$. It is straightforward to verify that $(T_1, <_1, \lambda)$ is a 1-consistent structure that satisfies G- and H-transfer. Hence we can apply Lemma 3. Thus there is a 1-SSM Σ_1 for $\Diamond_2\xi$ such that

$$\Sigma_1 = \{(\lambda_p, \lambda_q) : p, q \in U_1, (\exists p' \in T_1)(\exists q' \in T_1) p \equiv p' \ \& \ q \equiv q' \ \& \ p' <_1 q'\}$$

where $u \equiv v$ iff $\lambda_u = \lambda_v$ and $U_1 \subseteq T_1$ such that $|U_1| \leq 4 \times |\Gamma|$. Since every M_2^p is a 2-SSM, we get that (M_2^p, M_2^q) is indeed a 1-supermosaic for every $(\lambda_p, \lambda_q) \in \Sigma_1$, whence Σ_1 is a 1-SSS for ξ.

Finally, let us compute an upper bound on the size of Σ_1. Recall that $|U_1| \leq 4 \times |\Gamma|$, whence there are at most $(4 \times |\Gamma|)^2$ many 1-supermosaics in Σ_1. The size of the 1-supermosaics is also bounded by $(4 \times |\Gamma|)^2$, since $|U_2^p| \leq 4 \times |\Gamma|$. Thus the size of Σ_1 is bounded by $(4 \times |\Gamma|)^4$. □

6 Complexity

We are ready to provide a complete decision procedure in nondeterministic polynomial time for the satisfiability problem of the temporal logic of the lexicographic products of unbounded dense linear orders.

Theorem 1. *The satisfiability problem with respect to* $(\mathcal{C}_{ud}, \mathcal{C}_{ud})$ *is decidable in nondeterministic polynomial time.*

Proof. Given a formula ξ, let us proceed as follows.

1. Compute the least full domain Γ of formulas containing ξ; recall that $|\Gamma| \leq 14 \times (2 \times |\xi| + 10) + 2$.
2. nondeterministically choose a collection Σ_1 of 1-mosaics consisting of 2-mosaics of cardinality bounded by $(4 \times |\Gamma|)^4 \leq (4 \times 14 \times (2 \times |\xi| + 10) + 2)^4$.
3. Check whether Σ is indeed a 1-SSS for ξ.

By Lemma 2 and 4 the above decision procedure is complete. □

Recall that in Lemma 2 we constructed a model $(\mathbb{Q}, \mathbb{Q}, V)$ for ξ based on the rationals from a 1-SSS for ξ, the existence of which is equivalent to satisfiability of ξ by Lemma 4. Thus we have the following.

Theorem 2. *The logic of* $(\mathcal{C}_{ud}, \mathcal{C}_{ud})$ *coincides with the logic of the lexicographic product* $(\mathbb{Q}, \mathbb{Q})$ *of the rationals with the standard ordering.*

7 Conclusion

Temporal logics in which one can assign a proper meaning to the association of statements about different grained temporal domains have been considered. See [8, 12, 18] for details. Nevertheless, it seems that the results concerning the issues of axiomatization/completeness and decidability/complexity presented in [3] and in this paper constitute the first steps towards a temporal logic based on different levels of abstraction. Much remains to be done.

For example, one may consider the lexicographic products of special linear flows of time like $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$. Concerning the issues of axiomatization and completeness, could transfer results for completeness similar to the ones obtained by Kracht and Wolter [13] within the context of independently axiomatizable bimodal logics be obtained in our lexicographic setting? Concerning the issues of decidability and complexity, all normal extensions of $S4.3$, as proved in [6, 9], possess the finite model property and all finitely axiomatizable normal extensions of $K4.3$, as proved in [23], are decidable. Moreover, it follows from [15] that actually all finitely axiomatizable temporal logics of linear time flows are CoNP-complete. Is it possible to obtain similar results in our lexicographic setting? Or could undecidability results similar to the ones obtained by Reynolds and Zakharyaschev [21] within the context of the products of the modal logics determined by arbitrarily long linear orders be obtained in our lexicographic setting?

There is also the question of associating with $<_1$ and $<_2$ the until-like connectives U_1 and U_2 and the since-like connectives S_1 and S_2, the formulas $\varphi U_1 \psi$, $\varphi U_2 \psi$, $\varphi S_1 \psi$ and $\varphi S_2 \psi$ being read as one reads the formulas $\varphi U \psi$ and $\varphi S \psi$ in classical temporal logic, this time with $<_1$ and $<_2$. As yet, nothing has been done concerning the issues of axiomatization/completeness and decidability/complexity these new temporal connectives give rise to.

References

1. Balbiani, P.: Time representation and temporal reasoning from the perspective of non-standard analysis. In: Brewka, G., Lang, J. (eds.) Eleventh International Conference on Principles of Knowledge Representation and Reasoning, pp. 695–704. AAAI (2008)
2. Balbiani, P.: Axiomatization and completeness of lexicographic products of modal logics. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 165–180. Springer, Heidelberg (2009)
3. Balbiani, P.: Axiomatizing the temporal logic defined over the class of all lexicographic products of dense linear orders without endpoints. In: Markey, N., Wijsen, J. (eds.) Temporal Representation and Reasoning, pp. 19–26. IEEE (2010)
4. Van Benthem, J.: The Logic of Time. Kluwer (1991)
5. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press (2001)
6. Bull, R.: That all normal extensions of $S4.3$ have the finite model property. Zeitschrift für mathematische Logik und Grundlagen der Mathematik 12, 314–344 (1966)

7. Caleiro, C., Viganò, L., Volpe, M.: On the mosaic method for many-dimensional modal logics: a case study combining tense and modal operators. Logica Universalis 7, 33–69 (2013)
8. Euzenat, J., Montanari, A.: Time granularity. In: Fisher, M., Gabbay, D., Vila, L. (eds.) Handbook of Temporal Reasoning in Artificial Intelligence, pp. 59–118. Elsevier (2005)
9. Fine, K.: The logics containing $S4.3$. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik 17, 371–376 (1971)
10. Gabbay, D., Kurucz, A., Wolter, F., Zakharyaschev, M.: Many-Dimensional Modal Logics: Theory and Applications. Elsevier (2003)
11. Gabbay, D., Shehtman, V.: Products of modal logics, part 1. Logic Journal of the IGPL 6, 73–146 (1998)
12. Gagné, J.-R., Plaice, J.: A nonstandard temporal deductive database system. Journal of Symbolic Computation 22, 649–664 (1996)
13. Kracht, M., Wolter, F.: Properties of independently axiomatizable bimodal logics. Journal of Symbolic Logic 56, 1469–1485 (1991)
14. Kurucz, A.: Combining modal logics. In: Blackburn, P., van Benthem, J., Wolter, F. (eds.) Handbook of Modal Logic, pp. 869–924. Elsevier (2007)
15. Litak, T., Wolter, F.: All finitely axiomatizable tense logics of linear time flows are CoNP-complete. Studia Logica 81, 153–165 (2005)
16. Marx, M., Mikulás, S., Reynolds, M.: The mosaic method for temporal logics. In: Dyckhoff, R. (ed.) TABLEAUX 2000. LNCS (LNAI), vol. 1847, pp. 324–340. Springer, Heidelberg (2000)
17. Marx, M., Venema, Y.: Local variations on a loose theme: modal logic and decidability. In: Grädel, E., Kolaitis, P., Libkin, L., Marx, M., Spencer, J., Vardi, M., Venema, Y., Weinstein, S. (eds.) Finite Model Theory and its Applications, pp. 371–429. Springer (2007)
18. Nakamura, K., Fusaoka, A.: Reasoning about hybrid systems based on a nonstandard model. In: Orgun, M.A., Thornton, J. (eds.) AI 2007. LNCS (LNAI), vol. 4830, pp. 749–754. Springer, Heidelberg (2007)
19. Németi, I.: Decidable versions of first order logic and cylindric-relativized set algebras. In: Csirmaz, L., Gabbay, D., de Rijke, M. (eds.) Logic Colloquium 1992, pp. 171–241. CSLI Publications (1995)
20. Reynolds, M.: A decidable temporal logic of parallelism. Notre Dame Journal of Formal Logic 38, 419–436 (1997)
21. Reynolds, M., Zakharyaschev, M.: On the products of linear modal logics. Journal of Logic and Compution 11, 909–931 (2001)
22. Wolter, F.: Fusions of modal logics revisited. In: Kracht, M., de Rijke, M., Wansing, H., Zakharyaschev, M. (eds.) Advances in Modal Logic, pp. 361–379. CSLI Publications (1998)
23. Zakharyaschev, M., Alekseev, A.: All finitely axiomatizable normal extensions of $K4.3$ are decidable. Mathematical Logic Quarterly 41, 15–23 (1995)

Temporal Query Answering in the Description Logic *DL-Lite*[*]

Stefan Borgwardt, Marcel Lippmann, and Veronika Thost

Theoretical Computer Science, TU Dresden, Germany
{stefborg,lippmann,thost}@tcs.inf.tu-dresden.de

Abstract. Ontology-based data access (OBDA) generalizes query answering in relational databases. It allows to query a database by using the language of an ontology, abstracting from the actual relations of the database. For ontologies formulated in Description Logics of the *DL-Lite* family, OBDA can be realized by rewriting the query into a classical first-order query, e.g. an SQL query, by compiling the information of the ontology into the query. The query is then answered using classical database techniques.

In this paper, we consider a temporal version of OBDA. We propose a temporal query language that combines a linear temporal logic with queries over *DL-Lite*$_{core}$-ontologies. This language is well-suited to express temporal properties of dynamical systems and is useful in context-aware applications that need to detect specific situations. Using a first-order rewriting approach, we transform our temporal queries into queries over a temporal database. We then present three approaches to answering the resulting queries, all having different advantages and drawbacks.

1 Introduction

Context-aware applications try to detect specific situations within a changing environment (e.g. a computer system or air traffic observed by radar) to be able to react accordingly. To gain information, the environment is observed by sensors (for a computer system, data about its resources is gathered by the operating system), and the results of sensing are stored in a database. A context-aware application then detects specific predefined situations based on this data (e.g. a high system load) and reacts accordingly (e.g. by increasing the CPU frequency).

In a simple setting, such an application can be realized by using standard database techniques: the sensor information is stored in a database, and the situations to be recognized are specified as database queries [1]. In general, we cannot assume, however, that the sensors provide a complete description of the current state of the environment. For example, a sensor for certain information might not be available for a moment or not even exist. Thus, the closed world assumption employed by database systems (i.e. facts not present in the database are assumed to be false) is not appropriate since there may be facts of which it is unknown whether they hold or not.

[*] Partially supported by DFG SFB 912 (HAEC) and GRK 1763 (QuantLA).

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 165–180, 2013.

In addition, though a complete specification of the environment usually does not exist, often some knowledge about its behavior is available. This knowledge can be used to formulate constraints on the interpretation of the predicates used in the queries, to detect more complex situations. In ontology-based data access (OBDA) [10], domain knowledge is encoded in ontologies using a Description Logic (DL). In this paper, we consider logics of the *DL-Lite* family, which are light-weight DLs with a low complexity for many reasoning problems [10]. This low complexity is due to the fact that reasoning problems in *DL-Lite* can often be reduced to answering a first-order query over a relational database.

In order to recognize situations that evolve over time, we propose to add a temporal logical component to the queries. We use the operators of the temporal logic LTL, which allows to reason about a linear and discrete flow of time [20]. Usual temporal operators include *next* ($\bigcirc\phi$), which asserts that a property ϕ is true at the next point in time, *eventually* ($\Diamond\phi$), which asks for ϕ to be satisfied at some point in the future, and *always* ($\Box\phi$), which forces ϕ to be true at all time points in the future. We also use the corresponding past operators $\bigcirc^-$, $\Diamond^-$, and $\Box^-$.

Consider, for example, a collection of servers providing several services. An important task is to migrate services between servers to balance the load. To decide when to migrate, we want to detect certain critical situations. We consider a process to be critical if it has an increasing workload, and at the same time the server it is running on is almost overloaded. Suppose that we want to detect those processes and servers that were in a critical situation at least twice within the past ten time units. This can be expressed by the query $\bigcirc^{-10}(\Diamond(\mathsf{Critical}(x,y) \wedge \bigcirc\Diamond\mathsf{Critical}(x,y)))$, where

$$\begin{aligned}\mathsf{Critical}(x,y) := \mathsf{Server}(x) \wedge \mathsf{Process}(y) \wedge \mathsf{executes}(x,y) \wedge \mathsf{Running}(y) \wedge \\ \mathsf{IncreasingWorkload}(y) \wedge \mathsf{AlmostOverloaded}(x).\end{aligned}$$

In this example, it is essential that future and past operators can be nested arbitrarily. One might argue that, as we are looking at the time line from the point of view of the current time point, and nothing is known about the future, it is sufficient to have only past operators. We will even show that in our setting it is indeed always possible to construct an equivalent query using only past operators. However, the resulting query is not very concise and it is not easy to see the situation that is to be recognized. Indeed, for propositional LTL eliminating the past operators from a query results in a blowup that is at least exponential and no constructions of size less than triply exponential are known [18].

Temporal extensions of *DL-Lite* [11] have been considered in the context of conceptual modeling [2,3,4], where the focus lies on checking concept satisfiability instead of query answering. Investigations of temporalized OBDA, the second major use case of *DL-Lite*, with temporal query answering as the most important reasoning problem [10], have started only quite recently. In [16], a framework is developed that combines conjunctive queries in an arbitrary DL and the temporal logic LTL. The algorithm for query answering in this setting is an LTL-satisfiability test using a sub-procedure to answer (atemporal) CQs.

In [6], a similar query language, a combination of LTL and CQs over the DL $\mathcal{ALC}$, is proposed. In contrast to [16], its temporal component is allowed to influence the DL queries via the notion of rigid names, which are names whose interpretation does not change over time. The complexity increases depending on whether only rigid concept names or also rigid role names are allowed. Additionally, the latter paper also studies the so-called data complexity, where the complexity is measured only w.r.t. the size of the sensor data, i.e. the observations, but not w.r.t. the size of the query or the ontology. Another recent paper [5] examines temporal query answering in an extension of *DL-Lite* in which linear temporal operators are allowed to occur inside DL concepts, and proves first-order rewritability for query answering in this logic.

In this paper, we follow an approach suggested in [16] to combine the first-order rewriting techniques for atemporal query answering in logics of the *DL-Lite* family with a temporal component. The main idea is to use optimized database techniques to answer the actual queries. However, the existing techniques for answering temporal queries over temporal databases do not perfectly suit our purposes. In [14], the authors describe a temporal extension of the SQL query language that can answer temporal queries over a complete temporal database. However, in our setting the database containing all previous observations may grow huge very fast, but not all past observations are relevant for a particular query. In [13], an approach is described that reduces the amount of space needed; but the query language considered there allows only for past operators. In addition to describing how these approaches can be applied to our problem, we propose a new algorithm that extends the one from [13] and can also deal with future operators. All three approaches have different advantages and drawbacks.

Additionally, we show how the new algorithm can be extended to deal with rigid concept names for a specific subclass of queries. Unfortunately, there seems to be no simple way to adapt the algorithm to deal with rigid role names.

This paper is an extension of the recently appeared workshop paper [8]. The formal proofs of our results can be found in the technical report [9].

2 Preliminaries

We first describe the DL component, and then the temporal component of our query language. The *DL-Lite* family consists of various DLs that are tailored towards conceptual modeling and allow to realize query answering using classical database techniques. We only consider *DL-Lite*$_{core}$ as a prototypical example.

Definition 1. *Let* $\mathsf{N_C}$*,* $\mathsf{N_R}$*, and* $\mathsf{N_I}$ *be non-empty, pairwise disjoint sets of* concept, role*, and* individual names*, respectively. A* role expression *is either a role name* $P_1 \in \mathsf{N_R}$ *or an* inverse role P_2^- *with* $P_2 \in \mathsf{N_R}$*. A* basic concept *is of the form* A *or* $\exists R$*, where* $A \in \mathsf{N_C}$ *and* R *is a role expression. A* general concept *is of the form* B *or* $\neg B$*, where* B *is a basic concept.*

A concept inclusion *is of the form* $B \sqsubseteq C$*, where* B *is a basic concept and* C *is a general concept. An* assertion *is of the form* $A(a)$ *or* $P(a,b)$*, where* $A \in \mathsf{N_C}$*,*

$P \in \mathsf{N_R}$, and $a, b \in \mathsf{N_I}$. A TBox *(or* ontology*) is a finite set of concept inclusions, and an* ABox *is finite set of assertions.*

The semantics of *DL-Lite*$_{core}$ is defined through the notion of an interpretation.

Definition 2. *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 $A \in \mathsf{N_C}$ a set $A^\mathcal{I} \subseteq \Delta^\mathcal{I}$, to every $P \in \mathsf{N_R}$ a binary relation $P^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$, and to every $a \in \mathsf{N_I}$ an element $a^\mathcal{I} \in \Delta^\mathcal{I}$. This function is extended to role expressions, basic concepts, and general concepts as follows: $(P^-)^\mathcal{I} := \{(e,d) \mid (d,e) \in P^\mathcal{I}\}$, $(\exists R)^\mathcal{I} := \{d \mid$ there is an $e \in \Delta^\mathcal{I}$ such that $(d,e) \in R^\mathcal{I}\}$, and $(\neg C)^\mathcal{I} := \Delta^\mathcal{I} \setminus C^\mathcal{I}$.*

$\mathcal{I}$ is a model *of $B \sqsubseteq C$ if $B^\mathcal{I} \subseteq C^\mathcal{I}$, of $A(a)$ if $a^\mathcal{I} \in A^\mathcal{I}$, and of $P(a,b)$ if $(a^\mathcal{I}, b^\mathcal{I}) \in P^\mathcal{I}$. We write $\mathcal{I} \models \mathcal{T}$ if $\mathcal{I}$ is a model of all concept inclusions in the TBox $\mathcal{T}$, and $\mathcal{I} \models \mathcal{A}$ if $\mathcal{I}$ is a model of all assertions in the ABox $\mathcal{A}$. An ABox $\mathcal{A}$ is* consistent *(w.r.t. a TBox $\mathcal{T}$) if there is an $\mathcal{I}$ with $\mathcal{I} \models \mathcal{A}$ and $\mathcal{I} \models \mathcal{T}$.*

We assume that all interpretations $\mathcal{I}$ satisfy the *unique name assumption* (UNA), i.e. for all $a, b \in \mathsf{N_I}$ with $a \neq b$, we have $a^\mathcal{I} \neq b^\mathcal{I}$.

We now introduce the notion of temporal knowledge bases. Intuitively, they contain sensor data (ABoxes) for all previous time points, and a global TBox.

Definition 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 TBox $\mathcal{T}$, where the ABoxes $\mathcal{A}_i$ can only contain concept names that also occur in $\mathcal{T}$. Let $\mathfrak{I} = (\mathcal{I}_i)_{0 \leq i \leq n}$ be a sequence of interpretations $\mathcal{I}_i = (\Delta, \cdot^{\mathcal{I}_i})$ over a fixed non-empty domain Δ. Then $\mathfrak{I}$ is a* model *of $\mathcal{K}$ (written $\mathfrak{I} \models \mathcal{K}$) if $\mathcal{I}_i \models \mathcal{A}_i$ and $\mathcal{I}_i \models \mathcal{T}$ for all i, $0 \leq i \leq n$.*

Similar to what was done in [6,16], our temporal query language is based on conjunctive queries [1,12]. The main difference is that we do not allow for negation, as in *DL-Lite* arbitrary negation is disallowed. In contrast to [5], we also do not allow temporal operators inside concepts. These restrictions allow us to apply first-order rewritability of (atemporal) conjunctive queries in a black-box fashion to obtain a similar result for our temporal query language (see Section 3).

Definition 4. *Let $\mathsf{N_V}$ be a set of variables. A* conjunctive query *(CQ) is of the form $\phi = \exists y_1, \ldots, y_m.\psi$, where $y_1, \ldots, y_m \in \mathsf{N_V}$ and ψ is a (possibly empty) finite conjunction of* atoms *of the form $A(z)$ for $A \in \mathsf{N_C}$ and $z \in \mathsf{N_V} \cup \mathsf{N_I}$ (*concept atom*); or $r(z_1, z_2)$ for $r \in \mathsf{N_R}$ and $z_1, z_2 \in \mathsf{N_V} \cup \mathsf{N_I}$ (*role atom*). The empty conjunction is denoted by* `true`.

Temporal conjunctive queries *(TCQs) are built from CQs as follows: each CQ is a TCQ, and if ϕ_1 and ϕ_2 are TCQs, then so are $\phi_1 \wedge \phi_2$ (conjunction), $\phi_1 \vee \phi_2$ (disjunction), $\bigcirc\phi_1$ (strong next), $\bullet\phi_1$ (weak next), $\bigcirc^-\phi_1$ (strong previous), $\bullet^-\phi_1$ (weak previous), $\phi_1 \mathsf{U} \phi_2$ (until), and $\phi_1 \mathsf{S} \phi_2$ (since).*

The symbols $\bigcirc^-$, $\bullet^-$, and S are called *past operators*, the symbols $\bigcirc$, $\bullet$, and U are *future operators.* All results also hold in the presence of the additional temporal operators $\Box$ (always), $\Box^-$ (always in the past), $\Diamond$ (eventually), and $\Diamond^-$ (some time in the past) [9], but we omit them here for space reasons.

We denote the set of individuals occurring in a TCQ ϕ by $\mathsf{Ind}(\phi)$, the set of variables occurring in ϕ by $\mathsf{Var}(\phi)$, the set of free variables in ϕ by $\mathsf{FVar}(\phi)$, and the set of atoms occurring in ϕ by $\mathsf{At}(\phi)$. A TCQ ϕ is called *Boolean* if $\mathsf{FVar}(\phi) = \emptyset$. We further denote by $\mathsf{Sub}(\phi)$ the set of all TCQs occurring as subqueries in ϕ (including ϕ itself). A *union of conjunctive queries* (UCQ) is a disjunction of CQs. For our purposes, it is sufficient to define the semantics for Boolean CQs and TCQs. As usual, it is given using the notion of a homomorphism [12].

Definition 5. *Let $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ be an interpretation and ψ be a Boolean CQ. A mapping $\pi\colon \mathsf{Var}(\psi) \cup \mathsf{N_I} \to \Delta$ is a* homomorphism *of ψ into $\mathcal{I}$ if $\pi(a) = a^{\mathcal{I}}$ for all $a \in \mathsf{N_I}$, $\pi(z) \in A^{\mathcal{I}}$ 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 a sequence of interpretations $\mathfrak{I} = (\mathcal{I}_i)_{0 \le i \le n}$ and i with $0 \le i \le n$, we define $\mathfrak{I}, i \models \phi$ by induction on the structure of ϕ:

$$\begin{array}{ll}
\mathfrak{I}, i \models \exists y_1, \ldots, y_m.\psi & \textit{iff } \mathcal{I}_i \models \exists y_1, \ldots, y_m.\psi \\
\mathfrak{I}, i \models \phi_1 \land \phi_2 & \textit{iff } \mathfrak{I}, i \models \phi_1 \textit{ and } \mathfrak{I}, i \models \phi_2 \\
\mathfrak{I}, i \models \phi_1 \lor \phi_2 & \textit{iff } \mathfrak{I}, i \models \phi_1 \textit{ or } \mathfrak{I}, i \models \phi_2 \\
\mathfrak{I}, i \models \bigcirc\phi_1 & \textit{iff } i < n \textit{ and } \mathfrak{I}, i+1 \models \phi_1 \\
\mathfrak{I}, i \models \bullet\phi_1 & \textit{iff } i < n \textit{ implies } \mathfrak{I}, i+1 \models \phi_1 \\
\mathfrak{I}, i \models \bigcirc^-\phi_1 & \textit{iff } i > 0 \textit{ and } \mathfrak{I}, i-1 \models \phi_1 \\
\mathfrak{I}, i \models \bullet^-\phi_1 & \textit{iff } i > 0 \textit{ implies } \mathfrak{I}, i-1 \models \phi_1 \\
\mathfrak{I}, i \models \phi_1 \mathsf{U} \phi_2 & \textit{iff there is some } k,\ i \le k \le n \textit{ such that } \mathfrak{I}, k \models \phi_2 \\
 & \quad \textit{and } \mathfrak{I}, j \models \phi_1 \textit{ for all } j,\ i \le j < k \\
\mathfrak{I}, i \models \phi_1 \mathsf{S} \phi_2 & \textit{iff there is some } k,\ 0 \le k \le i \textit{ such that } \mathfrak{I}, k \models \phi_2 \\
 & \quad \textit{and } \mathfrak{I}, j \models \phi_1 \textit{ for all } j,\ k < j \le i.
\end{array}$$

Here we assume that there is no time point before 0 or after n, similar to the temporal semantics used for LTL in [23] or for temporal query languages for databases [13,17,21]. As in classical LTL, one can show that $\phi_1 \mathsf{S} \phi_2$ is equivalent to $\phi_2 \lor (\phi_1 \land \bigcirc^-(\phi_1 \mathsf{S} \phi_2))$, and a similar equivalence holds for U.

We are now ready to introduce the central reasoning problem of this paper, namely to find certain answers to TCQs.

Definition 6. *Let ϕ be a TCQ, $\mathfrak{I} = (\mathcal{I}_i)_{0 \le i \le n}$ a sequence of interpretations, and $i \ge 0$. The mapping $\mathfrak{a}\colon \mathsf{FVar}(\phi) \to \mathsf{N_I}$ is an* answer *to ϕ w.r.t. $\mathfrak{I}$ at time point i if $\mathfrak{I}, i \models \mathfrak{a}(\phi)$, where $\mathfrak{a}(\phi)$ denotes the Boolean TCQ that is obtained from ϕ by replacing the free variables according to $\mathfrak{a}$. Let further $\mathcal{K} = \langle (\mathcal{A}_i)_{0 \le i \le n}, \mathcal{T} \rangle$ be a TKB. A mapping $\mathfrak{a}\colon \mathsf{FVar}(\phi) \to \mathsf{N_I}$ is a* certain answer *to ϕ w.r.t. $\mathcal{K}$ at time point i if for every $\mathfrak{I} \models \mathcal{K}$, we have $\mathfrak{I}, i \models \mathfrak{a}(\phi)$.*

The set of all answers to ϕ w.r.t. $\mathfrak{I}$ at time point i is denoted by $\mathsf{Ans}(\phi, \mathfrak{I}, i)$, and the set of all certain answers to ϕ w.r.t. $\mathcal{K}$ is denoted by $\mathsf{Cert}(\phi, \mathcal{K}, i)$. Recall that our main interest lies in finding answers to queries at the current time point, i.e. computing the sets $\mathsf{Ans}(\phi, \mathfrak{I}) := \mathsf{Ans}(\phi, \mathfrak{I}, n)$ or $\mathsf{Cert}(\phi, \mathcal{K}) := \mathsf{Cert}(\phi, \mathcal{K}, n)$.

We will sometimes use the abbreviation $\mathsf{false} := A(x) \land A'(x)$, where A, A' are new concept names for which we assume that the concept inclusion $A \sqsubseteq \neg A'$ is contained in the global TBox $\mathcal{T}$.

3 Answering Temporal Conjunctive Queries

For computing the set of certain answers for a *conjunctive query*, the *rewriting approach* [10] can be employed. It compiles the information contained in the TBox into the query and evaluates the query w.r.t. the ABox (viewed as database) using classical database techniques. A similar approach is possible for TCQs.

Definition 7. *For an ABox $\mathcal{A}$, the interpretation* $\mathsf{DB}(\mathcal{A}) := (\mathsf{N_I}, \cdot^{\mathsf{DB}(\mathcal{A})})$ *is defined as follows:*

- $a^{\mathsf{DB}(\mathcal{A})} := a$ *for all* $a \in \mathsf{N_I}$*;*
- $A^{\mathsf{DB}(\mathcal{A})} := \{a \mid A(a) \in \mathcal{A}\}$ *for all* $A \in \mathsf{N_C}$*; and*
- $P^{\mathsf{DB}(\mathcal{A})} := \{(a,b) \mid P(a,b) \in \mathcal{A}\}$ *for all* $P \in \mathsf{N_R}$.

As shown in [10], this interpretation is the smallest model of $\mathcal{A}$. In order to employ database techniques, we must assume $\mathsf{DB}(\mathcal{A})$, and thus $\mathsf{N_I}$, to be finite.

Proposition 8 ([10]). *Let ψ be a CQ, $\mathcal{A}$ be an ABox, and $\mathcal{T}$ be a TBox. There is a* canonical model $\mathcal{I}_{\mathcal{A},\mathcal{T}}$ *of $\mathcal{A}$ and $\mathcal{T}$ and a UCQ $\psi^{\mathcal{T}}$ such that*

$$\mathsf{Cert}(\psi, \langle \mathcal{A}, \mathcal{T} \rangle) = \mathsf{Ans}(\psi, \mathcal{I}_{\mathcal{A},\mathcal{T}}) = \mathsf{Ans}(\psi^{\mathcal{T}}, \mathsf{DB}(\mathcal{A})).$$

We now use this proposition to show a similar result for TCQs. Let ϕ be a TCQ and $\mathcal{K} = \langle (\mathcal{A}_i)_{0 \leq i \leq n}, \mathcal{T} \rangle$ be a TKB. The TCQ $\phi^{\mathcal{T}}$ is obtained by replacing each CQ ψ occurring in ϕ by $\psi^{\mathcal{T}}$. Note that $\phi^{\mathcal{T}}$ is again a TCQ since $\psi^{\mathcal{T}}$ is always a UCQ. Let furthermore $\mathfrak{I}_{\mathcal{K}} := (\mathcal{I}_{\mathcal{A}_i,\mathcal{T}})_{0 \leq i \leq n}$ and $\mathsf{DB}(\mathcal{K}) := (\mathsf{DB}(\mathcal{A}_i))_{0 \leq i \leq n}$. The following theorem can be shown by a straightforward induction on the structure of ϕ.

Theorem 9. *For every TCQ ϕ, TKB $\mathcal{K} = \langle (\mathcal{A}_i)_{0 \leq i \leq n}, \mathcal{T} \rangle$, and $i \geq 0$, we have* $\mathsf{Cert}(\phi, \mathcal{K}, i) = \mathsf{Ans}(\phi, \mathfrak{I}_{\mathcal{K}}, i) = \mathsf{Ans}(\phi^{\mathcal{T}}, \mathsf{DB}(\mathcal{K}), i)$.

More importantly, for every TCQ ϕ and TKB $\mathcal{K} = \langle (\mathcal{A}_i)_{0 \leq i \leq n}, \mathcal{T} \rangle$, it holds that $\mathsf{Cert}(\phi, \mathcal{K}) = \mathsf{Ans}(\phi^{\mathcal{T}}, \mathsf{DB}(\mathcal{K}))$. It thus remains to show how to compute the set $\mathsf{Ans}(\phi, \mathfrak{I})$ for a TCQ ϕ and a sequence $\mathfrak{I} = (\mathcal{I}_i)_{0 \leq i \leq n}$ of interpretations over a finite domain. A first possibility is to view $\mathfrak{I}$ as a temporal database and rewrite ϕ into an ATSQL query [14]. However, since our goal is to monitor processes that produce new data in very short time intervals, storing all the data for all previous time points is not feasible. Therefore, we describe two different approaches that reduce the amount of space necessary to compute $\mathsf{Ans}(\phi, \mathfrak{I})$. Since we are interested in the answers at the last time point, the idea is to keep only the past information necessary to answer the query ϕ.

In the first approach (Section 4), we rewrite ϕ into a TCQ ϕ' without future operators, employing a construction described in [15]. We then compute $\mathsf{Ans}(\phi', \mathfrak{I})$ using an algorithm described in [13,22] that uses a so-called *bounded history encoding*, which means that the space required by the algorithm is constant w.r.t. the number n of previous time points. Only the current state of the database and some auxiliary relations have to be stored.

In Section 5, we generalize the algorithm from [13] to directly deal with the future operators. The main difference is that we do not consider negation or arbitrary first-order queries. Unfortunately, the space required by this algorithm is in general exponential in n and thus does not constitute a bounded history encoding in the sense of [13,22]. However, it allows us to circumvent the non-elementary blow-up of the formula resulting from the reduction in [15].

4 Eliminating Future Operators

To rewrite a TCQ ϕ into an equivalent TCQ that does not contain future operators, we employ the separation theorem for propositional LTL [15]. We describe here only the general idea, details can be found in the technical report [9].

The separation theorem cannot be applied directly since our temporal semantics differs from that in [15]: the only temporal operators in [15] are strict versions of U and S, and the semantics is defined w.r.t. bounded past and unbounded future. To apply this theorem, we replace the CQs in ϕ by propositional variables, rewrite U and S into their restrict counterparts, and use an additional propositional variable to delimit the time interval from 0 to n.

We can then apply the separation theorem to the resulting LTL-formula $\widehat{\phi}$. We obtain an equivalent LTL-formula $\widehat{\phi}'$ with negation which is a Boolean combination of temporal subformulae that either contain only strict S operators or only strict U operators. In this construction, subformulae of $\widehat{\phi}$ are copied and rearranged, but no additional propositional variables are introduced.

Since we are interested in evaluating ϕ at n, we can replace all variables in $\widehat{\phi}'$ that are in the scope of a strict U by false. The reason for this is that such variables are only evaluated at time points after n, where all variables are false. The resulting formula is then simplified to eliminate all strict U operators, and then translated back into a Boolean TCQ ϕ' by replacing the propositional variables by the corresponding CQs. Note that ϕ' contains no future operators.

We then apply the algorithm described in [13] to iteratively compute the answers to ϕ' at each time point.[1] The main advantage of this approach is that we can compute this set *iteratively* and such that the required memory is independent of the length of the sequence $\mathfrak{I}$. More formally, let $\mathfrak{I} = (\mathcal{I}_i)_{i \geq 0}$ be an *infinite* sequence of interpretations representing the observations over all time points. In our setting, these interpretations are generated from an infinite sequence of ABoxes that represent the observed sensor data using the construction of Section 3. At each time point $i \geq 0$, we only have access to the finite prefix $\mathfrak{I}^{(i)} := (\mathcal{I}_j)_{0 \leq j \leq i}$ of $\mathfrak{I}$ of length $i+1$. Let Δ be the shared domain of the interpretations in $\mathfrak{I}$.

The algorithm from [13] works on ϕ' as follows. On input $\mathcal{I}_0$, it computes a first-order interpretation $\mathcal{I}_0'$ of several auxiliary predicates. Intuitively, for each subformula ψ of ϕ' beginning with a past operator, the algorithm stores the answers $\mathsf{Ans}(\psi, \mathfrak{I}^{(0)}) \subseteq \Delta^{\mathsf{FVar}(\psi)}$ for ψ in a new relation $A_{\psi}^{\mathcal{I}_0'}$ of arity $|\mathsf{FVar}(\psi)|$.

[1] Before we can use the algorithm presented in [13], we need another rewriting step since in that paper the semantics of S is slightly different (see [9] for details).

The set $\mathsf{Ans}(\phi', \mathfrak{I}^{(0)})$ can then easily be computed from $\mathcal{I}_0$ and $\mathcal{I}'_0$. Afterwards, the algorithm disregards $\mathcal{I}_0$ and keeps only the information computed in $\mathcal{I}'_0$. On input $\mathcal{I}_1$, it then updates $\mathcal{I}'_0$ to a new interpretation $\mathcal{I}'_1$, which allows it to compute $\mathsf{Ans}(\phi', \mathfrak{I}^{(1)})$, and so on.

The memory requirements of this algorithm are bounded polynomially in the size of Δ, in the number of concept and role names, and in the number of past operators occurring in ϕ', and exponentially in the number of free variables occurring below past operators. However, the memory requirements do not depend on the length of the sequence of interpretations seen so far. This is called a *bounded history encoding* in [13].

Overall, the presented approach has, however, several drawbacks. First, the rewritings from ϕ to $\widehat{\phi}$ and from $\widehat{\phi'}$ to ϕ' may duplicate subformulae, which can cause exponential blowups in the size of ϕ. This could be avoided by applying a reduction similar to the one for propositional LTL in [15] directly to ϕ. However, since the reduction in [15] is already non-elementary in the size of the formula, this is not much more efficient. Hence, the presented approach is best suited for answering simple, small queries ϕ over large databases.

5 A New Algorithm

In this section, we present an algorithm that computes the set $\mathsf{Ans}(\phi, \mathfrak{I})$ without the need to eliminate the future operators beforehand, thereby avoiding the non-elementary blowup of the construction described in the previous section. However, the memory requirements of this new algorithm are not independent of the number of previous time points. From now on, let ϕ be a fixed TCQ and $\mathfrak{I} = (\mathcal{I}_i)_{i \geq 0}$ be a fixed infinite sequence of interpretations over the same finite domain Δ. For $i \geq 0$, we denote by $\mathfrak{I}^{(i)} := (\mathcal{I}_j)_{0 \leq j \leq i}$ the finite prefix of $\mathfrak{I}$ of length $i+1$. Our algorithm iteratively computes the sets $\mathsf{Ans}(\phi, \mathfrak{I}^{(i)})$. It uses as data structure so-called *answer formulae*, which represent TCQs in which some parts have already been evaluated. In particular, they do not contain CQs any more, but sets of already computed answers to subqueries. Additionally, they may contain variables (different from those in $\mathsf{N_V}$) that serve as place-holders for subqueries that have to be evaluated at the next time point.

For ease of presentation, we assume in the following that $\mathsf{N_V}$ is finite and that answers are of the form $\mathfrak{a} \colon \mathsf{N_V} \to \Delta$ instead of $\mathfrak{a} \colon \mathsf{FVar}(\phi) \to \Delta$. Thus, when we talk about answers, we mean mappings $\mathfrak{a} \colon \mathsf{N_V} \to \Delta$, and in particular $\mathsf{Ans}(\dots)$ refers to a set of such mappings, i.e. a subset of $\Delta^{\mathsf{N_V}}$.[2]

Definition 10. *Let $\mathsf{FSub}(\phi)$ denote the set of all subqueries of ϕ of the form $\bigcirc\psi_1$, $\CIRCLE\psi_1$, or $\psi_1 \mathbin{\mathsf{U}} \psi_2$. For $j \geq 0$, we denote by Var^{ϕ}_j the set of all variables of the form x^{ψ}_j for $\psi \in \mathsf{FSub}(\phi)$. The set AF^i_ϕ of all* answer formulae for ϕ at $i \geq 0$ *is the smallest set satisfying the following conditions:*

[2] In an implementation, one should restrict the intermediate computations of answers for subqueries ψ to $\mathsf{FVar}(\psi)$. But then one has to be more careful when combining answers to different subqueries.

Table 1. Computing answer formulae for a TCQ

ϕ	$\Phi_0(\phi)$	$\Phi_i^0(\phi)$
CQ ψ_1	$\mathsf{Ans}(\psi_1, \mathfrak{I}^{(0)})$	$\mathsf{Ans}(\psi_1, \mathfrak{I}^{(i)})$
$\psi_1 \wedge \psi_2$	$\Phi_0(\psi_1) \cap \Phi_0(\psi_2)$	$\Phi_i^0(\psi_1) \cap \Phi_i^0(\psi_2)$
$\psi_1 \vee \psi_2$	$\Phi_0(\psi_1) \cup \Phi_0(\psi_2)$	$\Phi_i^0(\psi_1) \cup \Phi_i^0(\psi_2)$
$\bigcirc\psi_1$	$x_0^{\bigcirc\psi_1}$	$x_i^{\bigcirc\psi_1}$
$\bigcirc^-\psi_1$	$\emptyset$	$\Phi_{i-1}(\psi_1)$
$\bullet\psi_1$	$x_0^{\bullet\psi_1}$	$x_i^{\bullet\psi_1}$
$\bullet^-\psi_1$	$\Delta^{\mathsf{N_V}}$	$\Phi_{i-1}(\psi_1)$
$\psi_1 \mathbin{\mathsf{U}} \psi_2$	$\Phi_0(\psi_2) \cup (\Phi_0(\psi_1) \cap x_0^{\psi_1 \mathsf{U} \psi_2})$	$\Phi_0(\psi_2) \cup (\Phi_i^0(\psi_1) \cap x_i^{\psi_1 \mathsf{U} \psi_2})$
$\psi_1 \mathbin{\mathsf{S}} \psi_2$	$\Phi_0(\psi_2)$	$\Phi_i^0(\psi_2) \cup (\Phi_i^0(\psi_1) \cap \Phi_{i-1}(\psi_1 \mathbin{\mathsf{S}} \psi_2))$

- *Every set* $A \subseteq \Delta^{\mathsf{N_V}}$ *is an answer formula for* ϕ *at* i*.*
- *Every* $x_j^\psi \in \mathsf{Var}_j^\phi$ *with* $j \leq i$ *is an answer formula for* ϕ *at* i*.*
- *If* α_1 *and* α_2 *are answer formulae for* ϕ *at* i*, then so are* $\alpha_1 \cap \alpha_2$ *and* $\alpha_1 \cup \alpha_2$*.*

In order to evaluate these answer formulae, we introduce the notion of correctness. Intuitively, an answer formula α for ϕ at i is correct for i if we obtain the set $\mathsf{Ans}(\phi, \mathfrak{I}^{(i)})$ by replacing the variables x_j^ψ in α by appropriate sets of answers and evaluating $\cap$ and $\cup$ as set intersection and union, respectively.

Definition 11. *We define the function* $\mathsf{eval}^n \colon \mathsf{AF}_\phi^n \to 2^{\Delta^{\mathsf{N_V}}}$, $n \geq 0$, *as follows:*

- $\mathsf{eval}^n(A) := A$ *if* $A \subseteq \Delta^{\mathsf{N_V}}$*;*
- $\mathsf{eval}^n(x_j^\psi) := \begin{cases} \mathsf{Ans}(\psi_1, \mathfrak{I}^{(n)}, j+1) & \text{if } j < n \text{ and } \psi = \bigcirc\psi_1 \text{ or } \psi = \bullet\psi_1; \\ \mathsf{Ans}(\psi, \mathfrak{I}^{(n)}, j+1) & \text{if } j < n \text{ and } \psi = \psi_1 \mathbin{\mathsf{U}} \psi_2; \\ \emptyset & \text{if } j = n \text{ and } \psi = \bigcirc\psi_1 \text{ or } \psi = \psi_1 \mathbin{\mathsf{U}} \psi_2; \\ \Delta^{\mathsf{N_V}} & \text{if } j = n \text{ and } \psi = \bullet\psi_1; \end{cases}$
- $\mathsf{eval}^n(\alpha_1 \cap \alpha_2) := \mathsf{eval}^n(\alpha_1) \cap \mathsf{eval}^n(\alpha_2)$*; and*
- $\mathsf{eval}^n(\alpha_1 \cup \alpha_2) := \mathsf{eval}^n(\alpha_1) \cup \mathsf{eval}^n(\alpha_2)$*.*

We say that a mapping $\Phi \colon \mathsf{Sub}(\phi) \to \mathsf{AF}_\phi^i$ *is* correct *for* $i \geq 0$ *if for all* $n \geq i$ *and for all* $\psi \in \mathsf{Sub}(\phi)$*, we have* $\mathsf{eval}^n(\Phi(\psi)) = \mathsf{Ans}(\psi, \mathfrak{I}^{(n)}, i)$*.*

In particular, if $\Phi \colon \mathsf{Sub}(\phi) \to \mathsf{AF}_\phi^i$ is correct for i, then $\mathsf{eval}^i(\Phi(\phi)) = \mathsf{Ans}(\phi, \mathfrak{I}^{(i)})$, which is the set we want to compute. Note that $x_j^{\psi_1 \mathsf{U} \psi_2}$ is actually a placeholder for $\bigcirc(\psi_1 \mathbin{\mathsf{U}} \psi_2)$ since we evaluate the U operator according to the recursive equivalence $\psi_1 \mathbin{\mathsf{U}} \psi_2 \equiv \psi_2 \vee (\psi_1 \wedge \bigcirc(\psi_1 \mathbin{\mathsf{U}} \psi_2))$ (cf. Table 1).

The algorithm works as follows. It first computes a mapping Φ_0 that is correct for 0, which is used to compute the next mapping Φ_1 when the interpretation $\mathcal{I}_1$ becomes available. This mapping is correct for 1 and can be used to compute the next mapping Φ_2, and so on. In each step, to compute Φ_{i+1}, we only need Φ_i and the interpretation $\mathcal{I}_{i+1}$. We recursively define the mapping $\Phi_0 \colon \mathsf{Sub}(\phi) \to \mathsf{AF}_\phi^0$ as shown in the second column of Table 1. Here, CQs are answered, e.g. by evaluating them as first-order queries over the database $\mathcal{I}_0$ [1].

Table 2. An example computation

ϕ	ψ_1	ψ_2	$\psi_1 \,\mathsf{S}\, \psi_2$
$\Phi_0(\phi)$	$x_0^{\psi_1}$	$B_0 \cup (A_0 \cap x_0^{\psi_2})$	$B_0 \cup (A_0 \cap x_0^{\psi_2})$
$\mathsf{Ans}(\phi, \mathfrak{I}^{(0)})$	$\Delta^{\mathsf{N_V}}$	B_0	B_0
$\Phi_1^0(\phi)$	$x_1^{\psi_1}$	$B_1 \cup (A_1 \cap x_1^{\psi_2})$	$\Phi_1^0(\psi_2) \cup (x_1^{\psi_1} \cap (B_0 \cup (A_0 \cap x_0^{\psi_2})))$
$\Phi_1(\phi)$	$x_1^{\psi_1}$	$B_1 \cup (A_1 \cap x_1^{\psi_2})$	$\Phi_1(\psi_2) \cup (x_1^{\psi_1} \cap (B_0 \cup (A_0 \cap \Phi_1(\psi_2))))$
			$\equiv ((x_1^{\psi_1} \cap B_0) \cup B_1) \cup (A_1 \cap x_1^{\psi_2})$
$\mathsf{Ans}(\phi, \mathfrak{I}^{(1)})$	$\Delta^{\mathsf{N_V}}$	B_1	$B_0 \cup B_1$
$\Phi_2(\phi)$	$x_2^{\psi_1}$	$B_2 \cup (A_2 \cap x_2^{\psi_2})$	$\Phi_2(\psi_2) \cup (x_2^{\psi_1} \cap (((x_1^{\psi_1} \cap B_0) \cup B_1) \cup (A_1 \cap x_1^{\psi_2})))$
			$\equiv ((x_2^{\psi_1} \cap ((C_1 \cap B_0) \cup B_1)) \cup B_2) \cup (A_1 \cap x_2^{\psi_2})$
$\mathsf{Ans}(\phi, \mathfrak{I}^{(2)})$	$\Delta^{\mathsf{N_V}}$	B_2	$(C_1 \cap B_0) \cup B_1 \cup B_2$

Assume now that $\Phi_{i-1} \colon \mathsf{Sub}(\phi) \to \mathsf{AF}_\phi^{i-1}$ is a function containing only variables with index $i-1$. We proceed as follows to construct a new function that contains only variables with index i. We recursively define the mapping $\Phi_i^0 \colon \mathsf{Sub}(\phi) \to \mathsf{AF}_\phi^i$ similarly to Φ_0 as given in the third column of Table 1.

Example 12. Consider the TCQ $\psi_1 \,\mathsf{S}\, \psi_2$ with two subqueries referring to the future, $\psi_1 := \bullet C(x)$ and $\psi_2 := A(x) \,\mathsf{U}\, B(x)$, and let $A_0 := \mathsf{Ans}(A(x), \mathfrak{I}^{(0)})$, and similarly for the other CQs and time points. The answer formulae Φ_0 and Φ_1^0 are listed in Table 2.

The difference to the definition of Φ_0 is that the answer formulae for past operators are computed using the answer formulae for the previous time point. This means that Φ_i^0 may still contain variables with index $i-1$. We now remove these old variables by substituting them appropriately. For example, since $x_{i-1}^{\bigcirc\psi}$ is a place-holder for the answers to ψ w.r.t. $\mathfrak{I}^{(n)}$ at i, we can now replace it by $\Phi_i^0(\psi)$. However, this formula may itself contain another old variable, and thus we have to be careful about the order in which we do these substitutions. Since each $\Phi_i^0(\psi)$ can contain only variables that refer to subqueries of ψ, by replacing the variables for "smaller" subqueries first, we ensure that all variables with index $i-1$ are eliminated. The details of this construction can be found in [9]. We obtain a mapping $\Phi_i \colon \mathsf{Sub}(\phi) \to \mathsf{AF}_\phi^i$ that is correct for i.

Lemma 13. *For each $i \geq 0$, the mapping Φ_i is correct for i.*

Example 14. Consider again the query $\psi_1 \,\mathsf{S}\, \psi_2$ from Example 12. Since $\Phi_1^0(\psi_1)$ and $\Phi_1^0(\psi_2)$ do not contain variables with index 0, the value of Φ_1 is the same as that of Φ_1^0 for both of these subqueries. We only have to replace $x_0^{\psi_2}$ within $\Phi_1^0(\psi_1 \,\mathsf{S}\, \psi_2)$ by $\Phi_1(\psi_2)$ to obtain $\Phi_1(\psi_1 \,\mathsf{S}\, \psi_2)$ as listed in Table 2. To obtain the answers at time point 1, we can now replace the remaining variables in according to eval^1, which yields $\mathsf{Ans}(\psi_1 \,\mathsf{S}\, \psi_2, \mathfrak{I}^{(1)}) = B_0 \cup B_1$.

Consider now the algorithm, which, on input ϕ and $\mathfrak{I}$, computes the mappings Φ_i as described above, and outputs $\mathsf{eval}^i(\Phi_i(\phi))$ for each $i \geq 0$. The following is a trivial consequence of the correctness of these mappings.

Theorem 15. *Given a TCQ* ϕ *and an infinite sequence* $\mathfrak{I} = (\mathcal{I}_i)_{i\geq 0}$ *of interpretations, the algorithm outputs* $\mathsf{Ans}(\phi, \mathfrak{I}^{(i)})$ *for each* $i \geq 0$.

It is easy to compute the sets $\mathsf{eval}^i(\Phi_i(\phi)) = \mathsf{Ans}(\phi, \mathfrak{I}^{(i)})$ for $i \geq 0$ since each of the variables x_i^ψ in $\Phi_i(\phi)$ simply has to be replaced by either $\emptyset$ or $\Delta^{\mathsf{N_V}}$ (see Definition 11). However, as mentioned earlier, the size of the formula $\Phi_i(\phi)$ may depend exponentially on the length i of the current sequence of interpretations.

Example 16. Consider again the query $\psi_1 \,\mathsf{S}\, \psi_2$ from Example 12. After replacing $x_0^{\psi_2}$ by $\Phi_1(\psi_2)$, the variable $x_1^{\psi_2}$ occurs twice in $\Phi_1(\psi_1 \,\mathsf{S}\, \psi_2)$. In general, $\Phi_i(\psi_1 \,\mathsf{S}\, \psi_2)$ will contain 2^i occurrences of the variable $x_i^{\psi_2}$. However, applying the associativity, commutativity, distributivity, and absorption laws for $\cap$ and $\cup$ does not affect the semantics of answer formulae (given by eval), and hence

$$\begin{aligned}
\Phi_1(\phi) &= \Phi_1(\psi_2) \cup (x_1^{\psi_1} \cap (B_0 \cup (A_0 \cap \Phi_1(\psi_2)))) \\
&\equiv \Phi_1(\psi_2) \cup (x_1^{\psi_1} \cap B_0) \cup (x_1^{\psi_1} \cap A_0 \cap \Phi_1(\psi_2)) \\
&\equiv \Phi_1(\psi_2) \cup (x_1^{\psi_1} \cap B_0) \\
&= (B_1 \cup (A_1 \cap x_1^{\psi_2})) \cup (x_1^{\psi_1} \cap B_0) \\
&\equiv ((x_1^{\psi_1} \cap B_0) \cup B_1) \cup (A_1 \cap x_1^{\psi_2})
\end{aligned}$$

The resulting formula contains $x_1^{\psi_2}$ only once. In general, the formula $\Phi_i(\phi)$ is equivalent to $((x_i^{\psi_1} \cap D_i) \cup B_i) \cup (A_i \cap x_i^{\psi_2})$, where $D_0 := \emptyset$ and for $i > 0$, we set $D_{i+1} := (C_{i+1} \cap D_i) \cup B_i$. Thus, the algorithm only has to store the sets $A_i, B_i, D_i \subseteq \Delta^{\mathsf{N_V}}$ at each time point, i.e. we achieve a bounded history encoding as in [13].

If the formula ϕ contains no future operators, then the answer formulae contain no variables and can always be fully evaluated to a subset of $\Delta^{\mathsf{N_V}}$. In this special case, our algorithm can be seen as a variant of the one from [13] for less expressive queries. Example 16 demonstrates that it is important that the computed answer formulae are simplified at each step, while preserving their semantics under eval. However, this does not guarantee a bounded history encoding as in [13].

6 Rigid Names

We now extend our temporal query language by designating certain concept names as being *rigid*, which means that their interpretation is not allowed to change over time. This especially makes sense regarding our application. For example, if the concept name Server describes the set of all severs, then it should be rigid since an application scenario with a server that stops being a server at some point in time would make no sense. The notion of rigidity has been explored for other temporal formalisms before [6,7].

For this purpose, we assume in this section that there is a set $\mathsf{N_{RC}} \subseteq \mathsf{N_C}$ of *rigid concept names*. In this setting, a finite sequence $\mathfrak{I} = (\mathcal{I}_i)_{0 \leq i \leq n}$ can only be a model of a TKB $\mathcal{K}$ if it fulfills the conditions of Definition 3 and additionally *respects* the rigid concept names, i.e. it satisfies $A^{\mathcal{I}_i} = A^{\mathcal{I}_j}$ for every rigid concept name A and all indices i, j between 0 and n.

For the remainder of this section, we restrict the query language to only allow so-called rooted CQs [19]. Intuitively, these are CQs that refer to at least one named individual.

Definition 17. *A CQ ϕ is called* rooted *if (i) it contains at least one free variable or individual name, and (ii) it is* connected*, i.e. for all $x, y \in \mathsf{Var}(\phi) \cup \mathsf{Ind}(\phi)$ there is a sequence $x_1, \dots, x_n \in \mathsf{Var}(\phi) \cup \mathsf{Ind}(\phi)$ such that $x_1 = x$, $x_n = y$, and for all i, $1 \leq i \leq n$, there is an $r \in \mathsf{N_R}$ such that either $r(x_i, x_{i+1}) \in \mathsf{At}(\phi)$ or $r(x_{i+1}, x_i) \in \mathsf{At}(\phi)$. A TCQ is* rooted *if it contains only rooted CQs.*

This makes sense from an application point of view since one usually does not ask if there is some object with certain properties, but actually wants to know the names of all objects with these properties. This restriction is not without loss of generality, but it is needed in the proof of Lemma 19. We have so far not been able to treat non-rooted TCQs in the presence of rigid concept names.

If we take the approach mentioned in Section 3 of viewing the input ABoxes as a temporal database and rewriting the TCQ into an ATSQL-query as in [14], then the additional rigidity constraints can simply be enforced by triggers that ensure that new knowledge about rigid names is added to the database at all previous time points.

However, the presence of rigid names poses a bigger problem for the incremental algorithm of [13] and that described in Section 5, both of which do not retain the data for all previous time points. For example, if the ABox at the next time point includes the assertion $A(a)$, where A is rigid, then this retroactively also changes the answers to the query $A(x)$ at previous time points. But the aforementioned algorithms assume that the answers at previous time points do not change.

Before we consider how to modify the algorithms for temporal query answering over databases, we have to show that we can still employ the rewriting approach and answer atemporal queries over a TKB $\mathcal{K}$ by directly querying the database $\mathsf{DB}(\mathcal{K})$. This means that we have to reconsider the proof of Theorem 9 regarding the interpretation of rigid names. The main problem we have to solve is that the sequence $\mathfrak{I}_{\mathcal{K}}$ of canonical models does not necessarily respect the rigid concept names. In the following, let $\mathcal{K} = \langle (\mathcal{A}_i)_{i \geq 0}, \mathcal{T} \rangle$ be an infinite TKB. Similar to Section 5, we denote by $\mathcal{K}^{(n)} := \langle (\mathcal{A}_i)_{0 \leq i \leq n}, \mathcal{T} \rangle$ the finite prefix of $\mathcal{K}$ of length $n + 1$. We show how to construct modified sequences of interpretations (similar to $\mathfrak{I}_{\mathcal{K}^{(n)}}$ from Theorem 9) that respect rigid names.

The first step is to find a set $\mathcal{R} \subseteq \{A(a) \mid A \in \mathsf{N_{RC}}, a \in \mathsf{N_I}\}$ that specifies the rigid concept names that the individual names are allowed to satisfy. Of course, we have to ensure that the assertions in $\mathcal{R}$ are not contradicted by any of the ABoxes $\mathcal{A}_i$, $i \geq 0$. We construct $\mathcal{R}$ iteratively, starting from $\mathcal{R}_0 := \emptyset$, as follows. In each step, we add to $\mathcal{R}_j$, $j \geq 0$, all assertions $A(a)$ with $A \in \mathsf{N_{RC}}$ and $a \in \mathsf{N_I}$

that are implied by $\mathcal{A}_i \cup \mathcal{R}_j$ w.r.t. $\mathcal{T}$ for some $i \geq 0$. This reasoning task is called *instance checking* and can be done in polynomial time in *DL-Lite*$_{core}$ [10]. This results in a new set $\mathcal{R}_{j+1}$. We iterate this process until no new assertions are added. Since there are only polynomially many assertions of the form $A(a)$ as above, this is possible in polynomial time. We denote by $\mathcal{R}$ the final set computed by this procedure. The next lemma shows that, in order to answer TCQs over $\mathcal{K}^{(n)}$, we can equivalently consider the TKB $\mathcal{K}^{(n)}_{\mathcal{R}} := \langle (\mathcal{A}_i \cup \mathcal{R})_{0\leq i\leq n}, \mathcal{T}\rangle$.

Lemma 18. *Let ϕ be a TCQ and $\mathcal{K} = \langle (\mathcal{A}_i)_{i\geq 0}, \mathcal{T}\rangle$ be an infinite TKB. Then there is a set $\mathcal{R}$ as above such that, for all i and n with $0 \leq i \leq n$, we have*

$$\mathsf{Cert}(\phi, \mathcal{K}^{(n)}, i) = \mathsf{Cert}(\phi, \mathcal{K}^{(n)}_{\mathcal{R}}, i).$$

Note that, if $\mathcal{R}$ is not consistent w.r.t. $\mathcal{T}$, this means that the TKB $\mathcal{K}$ is not consistent, i.e. there is no model of $\mathcal{K}$ that respects the rigid concept names.

Once we have computed $\mathcal{R}$, we can construct the desired sequence of canonical models that respects the rigid concept names, using an idea from [6]. We start with the original sequence $\mathfrak{I}_{\mathcal{K}^{(n)}_{\mathcal{R}}} = (\mathcal{I}_{\mathcal{A}_i\cup\mathcal{R},\mathcal{T}})_{0\leq i\leq n}$ that was used in Theorem 9 (but now with $\mathcal{K}^{(n)}_{\mathcal{R}}$ instead of $\mathcal{K}^{(n)}$). It is important to note that these canonical models, as constructed in [10], are all countable. We define the set $\mathcal{D} \subseteq 2^{\mathsf{N_{RC}}}$ of subsets of $\mathsf{N_{RC}}$ that contains exactly the sets

$$\rho(\mathcal{I}_{\mathcal{A}_i\cup\mathcal{R},\mathcal{T}}, x) := \{A \in \mathsf{N_{RC}} \mid x \in A^{\mathcal{I}_{\mathcal{A}_i\cup\mathcal{R},\mathcal{T}}}\}$$

for all i, $0 \leq i \leq n$, and $x \in \Delta^{\mathcal{I}_{\mathcal{A}_i\cup\mathcal{R},\mathcal{T}}}$. We will now modify each $\mathcal{I}_{\mathcal{A}_i\cup\mathcal{R},\mathcal{T}}$ into a new interpretation $\mathcal{I}_i$ such that for each $Y \in \mathcal{D}$ there are countably infinitely many individuals $x \in \Delta^{\mathcal{I}_i}$ with $Y = \rho(\mathcal{I}_i, x)$.

To this end, consider i, n, $0 \leq i \leq n$, and $Y \in \mathcal{D}$. If $\mathcal{I}_{\mathcal{A}_i\cup\mathcal{R},\mathcal{T}}$ does not contain any such individual, then we first have to add one. Fortunately, from the definition of $\mathcal{D}$ we know that there must be a j, $0 \leq j \leq n$, and $x \in \Delta^{\mathcal{I}_{\mathcal{A}_j\cup\mathcal{R},\mathcal{T}}}$ such that $Y = \rho(\mathcal{I}_{\mathcal{A}_j\cup\mathcal{R},\mathcal{T}}, x)$. To be on the safe side, we therefore construct the disjoint union $\mathcal{I}'_i$ of all interpretations in $\mathfrak{I}_{\mathcal{K}^{(n)}_{\mathcal{R}}}$. More formally, the domain of $\mathcal{I}'_i$ is the disjoint union of the domains of $\mathcal{I}_{\mathcal{A}_j\cup\mathcal{R},\mathcal{T}}$, $0 \leq j \leq n$. The concept and role names are interpreted as the (disjoint!) union of the interpretations of these names under all $\mathcal{I}_{\mathcal{A}_j\cup\mathcal{R},\mathcal{T}}$, while the individual names are interpreted as in $\mathcal{I}_{\mathcal{A}_i\cup\mathcal{R},\mathcal{T}}$. Note that the components of $\mathcal{I}'_i$ are not connected by any roles. This implies in particular that any homomorphism of a *rooted* CQ into $\mathcal{I}'_i$ must actually be a homomorphism into the original canonical model $\mathcal{I}_{\mathcal{A}_i\cup\mathcal{R},\mathcal{T}}$. This fact is essential to prove Lemma 19 below (see [9] for details).

To ensure that there are even countably infinitely many such individuals, we now define $\mathcal{I}''_i$ as the countably infinite disjoint union of $\mathcal{I}'_i$ with itself, where again the interpretation of the individual names remains unchanged. Finally, we ensure that all models have the same domain $\Delta := \mathsf{N_I} \cup (\mathcal{D} \times \mathbb{N})$ and interpret the individual names by the same domain elements by applying a bijection between the domain of each $\mathcal{I}''_i$ and Δ. In particular, each $a^{\mathcal{I}''_i}$ for $a \in \mathsf{N_I}$ is simply mapped to a, and every other element $x \in \Delta^{\mathcal{I}''_i}$ is mapped to some $(\rho(\mathcal{I}''_i, x), \ell)$ with $\ell \in \mathbb{N}$. We denote the resulting interpretation by $\mathcal{I}_i$ and define $\mathfrak{I}_{\mathcal{K}^{(n)},\mathcal{R}} := (\mathcal{I}_i)_{0\leq i\leq n}$.

Algorithm 1. Compute certain answers to a rooted TCQ w.r.t. rigid names

```
Input   : A rooted TCQ $\phi$ and an infinite TKB $\mathcal{K} = \langle(\mathcal{A}_i)_{i\geq 0}, \mathcal{T}\rangle$
Output  : $\mathsf{Cert}(\phi, \mathcal{K}^{(i)})$ for each $i \geq 0$

for $\mathcal{R} \in \mathfrak{R}$ do
    initialize an instance $\mathsf{A}_\mathcal{R}$ of the algorithm of Section 5 with $\phi^\mathcal{T}$
end
for $i \leftarrow 0, 1, \dots$ do
    for $\mathcal{R} \in \mathfrak{R}$ do
        run $\mathsf{A}_\mathcal{R}$ on input $\mathsf{DB}(\mathcal{A}_i \cup \mathcal{R})$ to compute $\mathsf{Cert}(\phi, \mathcal{K}_\mathcal{R}^{(i)})$
    end
    output $\bigcap_{\mathcal{R} \in \mathsf{Active}} \mathsf{Cert}(\phi, \mathcal{K}_\mathcal{R}^{(i)})$
end
```

Lemma 19. *The sequence $\mathfrak{I}_{\mathcal{K}^{(n)},\mathcal{R}}$ is a model of $\mathcal{K}_\mathcal{R}^{(n)}$. Furthermore, for all rooted CQs ϕ and every i, $0 \leq i \leq n$, we have*

$$\mathsf{Ans}(\phi, \mathfrak{I}_{\mathcal{K}^{(n)},\mathcal{R}}, i) = \mathsf{Ans}(\phi, \mathfrak{I}_{\mathcal{K}_\mathcal{R}^{(n)}}, i).$$

We can now finally state the variant of Theorem 9 that can deal with rigid concept names.

Theorem 20. *Let ϕ be a rooted TCQ and $\mathcal{K} = \langle(\mathcal{A}_i)_{i\geq 0}, \mathcal{T}\rangle$ be an infinite TKB. Then there is a set $\mathcal{R}$ as above such that, for all i and n with $0 \leq i \leq n$, we have*

$$\mathsf{Cert}(\phi, \mathcal{K}_\mathcal{R}^{(n)}, i) = \mathsf{Ans}(\phi, \mathfrak{I}_{\mathcal{K}^{(n)},\mathcal{R}}, i) = \mathsf{Ans}(\phi^\mathcal{T}, \mathsf{DB}(\mathcal{K}_\mathcal{R}^{(n)}), i).$$

Note that $\mathsf{DB}(\mathcal{K}_\mathcal{R}^{(n)})$ is independent of the construction of $\mathfrak{I}_{\mathcal{K}^{(n)},\mathcal{R}}$, and we can now simply apply the algorithm of Section 5 to the modified sequence of interpretations $\mathsf{DB}(\mathcal{K}_\mathcal{R}^{(n)})$ instead of $\mathsf{DB}(\mathcal{K}^{(n)})$. More formally, let $\mathfrak{R}$ denote the set of all sets $\mathcal{R}$ of the form described above. Algorithm 1 describes the steps necessary to compute the certain answers to a TCQ in the presence of rigid names.

For each $\mathcal{R} \in \mathfrak{R}$, we start an instance $\mathsf{A}_\mathcal{R}$ of the algorithm presented in Section 5. All these instances are run in parallel, with the only difference between them being that each instance has a fixed set $\mathcal{R}$ of assumptions about the rigid names. In each step, every instance $\mathsf{A}_\mathcal{R}$ computes the certain answers to ϕ relative to $\mathcal{R}$, and the actual set of certain answers to ϕ is then computed by taking the intersection over all these sets. Note that we could in each step terminate those instances $\mathsf{A}_\mathcal{R}$ for which $\mathcal{A}_i \cup \mathcal{R}$ is inconsistent w.r.t. $\mathcal{T}$ since this implies that $\mathcal{K}_\mathcal{R}^{(i)}$ has no models, and thus $\mathsf{Cert}(\phi, \mathcal{K}_\mathcal{R}^{(i)}) = \Delta^{\mathsf{N_V}}$ does not contribute to the computation of the intersection in Algorithm 1. However, we leave out this simple optimization to make the presentation of the algorithm clearer.

Theorem 21. *Given a rooted TCQ ϕ and an infinite TKB $\mathcal{K} = \langle(\mathcal{A}_i)_{i\geq 0}, \mathcal{T}\rangle$, Algorithm 1 outputs $\mathsf{Cert}(\phi, \mathcal{K}^{(i)})$ for each $i \geq 0$.*

We have thus extended the algorithm in Section 5—and by extension also the algorithm described in [13]—to deal with rigid concept names in rooted TCQs.

7 Conclusions

We have introduced the reasoning task of *temporal OBDA* over *DL-Lite* knowledge bases and shown how to reduce this task to answering queries over temporal databases, similar to what was done for the atemporal case [10]. We then presented three approaches to solve the latter problem. The first involves storing the whole history of the database and re-evaluating the query at each time point using a temporal database query language like ATSQL [14].

The second approach works by eliminating the future operators and evaluating the resulting query using the algorithm of [13], which achieves a bounded history encoding. Although independent of the length of the history, this involves a non-elementary blow-up in the size of the query. Then, we presented an algorithm that works directly with the future operators. We showed that the algorithm computes exactly the desired answers, but its space requirements are in general not independent of the length of the history. In future work, we will try to achieve a bounded history encoding for certain classes of TCQs, and compare the performance of all three approaches on temporal databases.

Finally, we also described an approach to extend the proposed algorithm to deal with rigid concept names if only rooted CQs are allowed. We plan to investigate how to adapt the algorithm to deal also with rigid role names.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Artale, A., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: *DL-Lite* with temporalised concepts, rigid axioms and roles. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 133–148. Springer, Heidelberg (2009)
3. Artale, A., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: Temporal conceptual modelling with DL-Lite. In: Proc. of the 2010 Int. Workshop on Description Logics (DL 2010). CEUR Workshop Proceedings, vol. 573. CEUR-WS.org (2010)
4. Artale, A., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: A cookbook for temporal conceptual data modelling with description logics. CoRR abs/1209.5571 (2012), http://arxiv.org/abs/1209.5571
5. Artale, A., Kontchakov, R., Wolter, F., Zakharyaschev, M.: Temporal description logic for ontology-based data access. In: Proc. of the 23rd Int. Joint Conf. on Artificial Intelligence (IJCAI 2013). AAAI Press (2013)
6. Baader, F., Borgwardt, S., Lippmann, M.: Temporalizing ontology-based data access. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 330–344. Springer, Heidelberg (2013)
7. Baader, F., Ghilardi, S., Lutz, C.: LTL over description logic axioms. ACM Transactions on Computational Logic 13(3), 21:1–21:32 (2012)

8. Borgwardt, S., Lippmann, M., Thost, V.: Temporal query answering in *DL-Lite*. In: Proc. of the 26th Int. Workshop on Description Logics (DL 2013). CEUR Workshop Proceedings. CEUR-WS.org (2013)
9. Borgwardt, S., Lippmann, M., Thost, V.: Temporal query answering w.r.t. *DL-Lite*-ontologies. LTCS-Report 13-05, Chair of Automata Theory, TU Dresden, Dresden, Germany (2013), see, `http://lat.inf.tu-dresden.de/research/reports.html`
10. 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 2009. LNCS, vol. 5689, pp. 255–356. Springer, Heidelberg (2009)
11. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: DL-Lite: Tractable description logics for ontologies. In: Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI 2005), pp. 602–607. AAAI Press (2005)
12. Chandra, A.K., Merlin, P.M.: Optimal implementation of conjunctive queries in relational data bases. In: Hopcroft, J.E., Friedman, E.P., Harrison, M.A. (eds.) Proc. of the 9th Annual ACM Symp. on Theory of Computing (STOC 1977), pp. 77–90. ACM Press (1977)
13. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Transactions on Database Systems 20(2), 148–186 (1995)
14. Chomicki, J., Toman, D., Böhlen, M.H.: Querying ATSQL databases with temporal logic. ACM Transactions on Database Systems 26(2), 145–178 (2001)
15. Gabbay, D.: The declarative past and imperative future. In: Banieqbal, B., Barringer, H., Pnueli, A. (eds.) Temporal Logic in Specification. LNCS, vol. 398, pp. 409–448. Springer, Heidelberg (1989)
16. Gutiérrez-Basulto, V., Klarman, S.: Towards a unifying approach to representing and querying temporal data in description logics. In: Krötzsch, M., Straccia, U. (eds.) RR 2012. LNCS, vol. 7497, pp. 90–105. Springer, Heidelberg (2012)
17. Hülsmann, K., Saake, G.: Theoretical foundations of handling large substitution sets in temporal integrity monitoring. Acta Informatica 28(4), 365–407 (1991)
18. Laroussinie, F., Markey, N., Schnoebelen, P.: Temporal logic with forgettable past. In: Proc. of the 17th Annual IEEE Symp. on Logic in Computer Science (LICS 2002), pp. 383–392. IEEE Press (2002)
19. 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)
20. Pnueli, A.: The temporal logic of programs. In: Proc. of the 18th Annual Symp. on Foundations of Computer Science (SFCS 1977), pp. 46–57 (1977)
21. Saake, G., Lipeck, U.W.: Using finite-linear temporal logic for specifying database dynamics. In: Börger, E., Büning, H.K., Richter, M.M. (eds.) CSL 1988. LNCS, vol. 385, pp. 288–300. Springer, Heidelberg (1989)
22. Toman, D.: Logical data expiration. In: Chomicki, J., van der Meyden, R., Saake, G. (eds.) Logics for Emerging Applications of Databases, ch. 6, pp. 203–238. Springer (2004)
23. Wilke, T.: Classifying discrete temporal properties. In: Meinel, C., Tison, S. (eds.) STACS 1999. LNCS, vol. 1563, pp. 32–46. Springer, Heidelberg (1999)

Verification of Golog Programs over Description Logic Actions

Franz Baader and Benjamin Zarrieß*

Theoretical Computer Science, TU Dresden, Germany
{baader,zarriess}@tcs.inf.tu-dresden.de

Abstract. High-level action programming languages such as Golog have successfully been used to model the behavior of autonomous agents. In addition to a logic-based action formalism for describing the environment and the effects of basic actions, they enable the construction of complex actions using typical programming language constructs. To ensure that the execution of such complex actions leads to the desired behavior of the agent, one needs to specify the required properties in a formal way, and then verify that these requirements are met by any execution of the program. Due to the expressiveness of the action formalism underlying Golog (Situation Calculus), the verification problem for Golog programs is in general undecidable. Action formalisms based on Description Logic (DL) try to achieve decidability of inference problems such as the projection problem by restricting the expressiveness of the underlying base logic. However, until now these formalisms have not been used within Golog programs. In the present paper, we introduce a variant of Golog where basic actions are defined using such a DL-based formalism, and show that the verification problem for such programs is decidable. This improves on our previous work on verifying properties of infinite sequences of DL actions in that it considers (finite and infinite) sequences of DL actions that correspond to (terminating and non-terminating) runs of a Golog program rather than just infinite sequences accepted by a Büchi automaton abstracting the program.

1 Introduction

Action programming languages like Golog [8,11] can be used to control the behavior of autonomous agents and mobile robots. In this setting, the programming language provides the user with typical programming language constructs such as loops and tests. These constructs can be used to build complex actions from atomic ones. The semantics of the atomic actions and of the programming language constructs is formally specified using an appropriate logical calculus (Situation Calculus in the case of Golog). To ensure that a complex action specified in this way actually shows the desired behavior, one needs to specify the required properties in a formal way, and then verify that these requirements are met by any run of the program defining this action. In principle, this verification

* Supported by DFG Research Unit FOR 1513, project A1.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 181–196, 2013.

problem boils down to a deduction problem in the underlying logical calculus, but due to the expressiveness of the situation calculus, the deduction problem is in general undecidable. For instance, the first work that aims at the fully automated verification of (non-terminating) Golog programs [7] specifies properties in an extension of the situation calculus by constructs of the temporal logic CTL*. Similar to CTL* model checking, the approach tries to compute an appropriate fixpoint, but in contrast to the case of model checking the fixpoint iteration need not terminate. If it does terminates, then the proof that the desired property holds is reduced to a deduction problem in the underlying logic (i.e., situation calculus), which is in general not decidable.

In order to overcome the problems caused by an undecidable base logic, action theories based on decidable Description Logics [2] have been proposed in the literature [1,10,4]. The decidability and complexity results obtained in these papers were mainly concerned with the projection problem: given a finite sequence of atomic actions and a (possibly incomplete) description of the initial world, decide whether a certain property is guaranteed to hold after the execution of this sequence. The papers differ w.r.t. what kind of domain constraints (i.e., constraints that are guaranteed to hold in every world) can be used. Whereas [1] restricts the attention to acyclic TBoxes, the other two papers allow for general TBoxes, i.e., finite sets of general concept inclusions (GCIs). In the presence of GCIs, one has to deal with the so-called ramification problem, i.e., the fact that the direct effects of an action may violate the domain constraints, and thus also indirect effects need to be considered. Whereas the authors of [10] deal with this problem by introducing so-called occlusions, the approach described in [4] uses so-called causal relationships to specify indirect effects of actions.

The first attempt to extend the decidability results for projection in [1] to the verification problem can be found in [5]. However, instead of examining the actual execution sequences of a given Golog program, this approach considers infinite sequences of actions that are accepted by a given Büchi automaton $\mathcal{B}$. If $\mathcal{B}$ is an upper approximation of the program, i.e. all possible execution sequences of the program are accepted by $\mathcal{B}$, then any property that holds in all the sequences accepted by $\mathcal{B}$ is also a property that is satisfied by any execution of the program. As logic for specifying properties of infinite sequences of DL actions, the approach uses the temporalized DL $\mathcal{ALC}$-LTL [3], which extends the well-known propositional linear temporal logic (LTL) [12] by allowing for the use of axioms (i.e., TBox and ABox statements) of the basic DL $\mathcal{ALC}$ in place of propositional letters.[1] Recently, other restrictions on action theories based on the situation calculus that guarantee decidability of the verification problem were considered [9]. However, instead of considering execution sequences of programs, this work looks at all possible infinite sequences of actions. For a more detailed discussion of related work see [6].

In the present paper, we improve on the results in [5] in several respects. First, instead of using Büchi automata to approximate programs, we directly consider

[1] More precisely, [5] uses the extension of $\mathcal{ALC}$-LTL to the more expressive DL $\mathcal{ALCO}$, but disallows TBox statements.

Golog programs. Second, we deal with terminating and non-terminating runs of programs in a uniform way. Finally, our approach works not only for the action formalism introduced in [1], but also for the one considered in [4]. Regarding the underlying DL, our result applies to all DLs considered in [1], but for the sake of simplicity we restrict the attention to the DL $\mathcal{ALCO}$.

In the next section, we introduce the relevant notions concerning DL and action languages based on DLs. Then we define syntax and semantics of Golog programs over DL actions and prove some auxiliary results for such programs. In the subsequent section, we define the verification problem and show that it is decidable. Because of space constraints, detailed proofs of our results have to be omitted. They can be found in [6].

2 Preliminaries

Description Logics. The DL $\mathcal{ALCO}$ extends the basic DL $\mathcal{ALC}$ by nominals, i.e., singleton concepts. Starting with sets N_C of *concept names*, N_R of *role names*, and N_I of *individual names*, $\mathcal{ALCO}$-concept descriptions (*concepts* for short) are built from concept names using the *constructors* shown in Table 1.

Table 1. Syntax and semantics of $\mathcal{ALCO}$

DL	Name	Syntax	Semantics
$\mathcal{ALC}$	top-concept	$\top$	$\Delta^\mathcal{I}$
	negation	$\neg C$	$\Delta^\mathcal{I} \setminus C^\mathcal{I}$
	conjunction	$C \sqcap D$	$C^\mathcal{I} \cap D^\mathcal{I}$
	disjunction	$C \sqcup D$	$C^\mathcal{I} \cup D^\mathcal{I}$
	existential restriction	$\exists r.C$	$\{x \mid \exists y : (x,y) \in r^\mathcal{I} \wedge y \in C^\mathcal{I}\}$
	value restriction	$\forall r.C$	$\{x \mid \forall y : (x,y) \in r^\mathcal{I} \rightarrow y \in C^\mathcal{I}\}$
$\mathcal{O}$	nominal	$\{a\}$	$\{a^\mathcal{I}\}$

In the following, we often use A, B to denote concept names, r, s for role names, a, b for individual names (*individuals* for short), and C, D for possibly complex concepts.

An *ABox* is a finite set of *concept assertions* $C(a)$ and positive and negated *role assertions* of the form $r(a,b)$ and $\neg r(a,b)$, respectively. Assertions of the form $A(a), \neg A(a), r(a,b), \neg r(a,b)$ for concept names A and role names r are called *literals.*

A *concept definition* is of the form $A \equiv C$ and a *general concept inclusion* (GCI) is of the form $C \sqsubseteq D$. An *acyclic TBox* is a finite set of concept definitions with unique left-hand sides. Additionally, it is required that there are no cyclic dependencies between the definitions. A *general* TBox is a finite set of GCIs.

The semantics of concepts is defined in terms of interpretations. An *interpretation* $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ consists of a non-empty domain $\Delta^\mathcal{I}$ and a mapping $\cdot^\mathcal{I}$, which maps each concept name A to a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, each role name r to a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and each individual a to an element $a^{I} \in \Delta^{I}$. We assume that $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ for any two distinct individuals a, b (*unique name assumption*). The extension of $\cdot^{\mathcal{I}}$ to complex concepts is defined inductively as shown in Table 1.

An interpretation $\mathcal{I}$ satisfies an ABox assertion $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$, $r(a, b)$ if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$, and $\neg r(a, b)$ if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \notin r^{\mathcal{I}}$. It is a *model* of an ABox $\mathcal{A}$ (written as $\mathcal{I} \models \mathcal{A}$) if $\mathcal{I}$ satisfies all assertions in $\mathcal{A}$. An interpretation $\mathcal{I}$ satisfies a concept definition $A \equiv C$ if $A^{\mathcal{I}} = C^{\mathcal{I}}$ and a GCI $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. It is a model of a TBox $\mathcal{T}$ (written $\mathcal{I} \models \mathcal{T}$) if it satisfies each definition or GCI, respectively, in $\mathcal{T}$. The ABox $\mathcal{A}$ is *consistent* w.r.t. the TBox $\mathcal{T}$ if there exists a model of $\mathcal{A}$ that is also a model of $\mathcal{T}$. The set of models of $\mathcal{A}$ and $\mathcal{T}$ is denoted by $\mathcal{M}(\mathcal{A})$ and $\mathcal{M}(\mathcal{T})$, respectively. The assertion φ is *entailed* by $\mathcal{A}$ and $\mathcal{T}$ (written as $\mathcal{T}, \mathcal{A} \models \varphi$) if every model of $\mathcal{A}$ and $\mathcal{T}$ (i.e., element of $\mathcal{M}(\mathcal{A}) \cap \mathcal{M}(\mathcal{T})$) satisfies φ.

For $\mathcal{ALCO}$, the consistency and the entailment problem are PSpace-complete w.r.t. an acyclic TBox and ExpTime-complete w.r.t. a general TBox [2].

DL-Based Action Formalism. Instead of introducing a specific DL-based action formalism, we take a more abstract point of view and describe a whole class of DL-based action formalisms. This class has the formalisms introduced in [1] (without occlusions) and in [4] as instances, but not the one of [10] (since there occlusions are a key ingredient of the formalism). Basically, we abstract from the concrete way the formalism determines the effect of applying an action to a world, and assume that there is an appropriate function that provides us with the effect. Later on, we need to impose additional restrictions in order to obtain our decidability results.

A *DL-based action theory* consists of the following components:

– the *domain constraints*, given as a TBox $\mathcal{T}$;
– an incomplete description of the *initial world* given by an ABox $\mathcal{A}$;
– a finite set Σ of *action names*;
– a finite set of *relevant ABox assertions*, denoted by $\mathcal{D}$.

In the following we use the (possibly indexed) letters α, β to denote action names and φ to denote ABox assertions (or assertions for short). The set of literals contained in $\mathcal{D}$ is denoted by *Lit*. We require that the assertions contained in the description of the initial world are also contained in $\mathcal{D}$, and that $\mathcal{D}$ is closed under negation (modulo elimination of double negation). For the formalism in [1], the elements of $\mathcal{D}$ are the assertions occurring in the initial ABox and in the pre- and post-conditions of action descriptions.

Instead of introducing the syntactic form of action descriptions and then defining the effects of actions by providing these descriptions with an appropriate semantics, we define the semantics of action names directly using an effect function. The literals in $\mathcal{D}$ are used to specify how an action changes the actual world. An interpretation $\mathcal{I}$ completely describes the current state of the world. The semantics of actions is thus defined by specifying how they transform a given interpretation into a successor interpretation. First, we have to determine whether an action α is *applicable* to an interpretation $\mathcal{I}$. Then, if α is applicable to $\mathcal{I}$, we define the *effects* of α on $\mathcal{I}$ as a set of literals.

Definition 1. *Let Σ be the set of action names, $\mathcal{D}$ the set of relevant assertions with $Lit \subseteq \mathcal{D}$ the set of literals occurring in $\mathcal{D}$, and $\mathcal{T}$ the TBox specifying the domain constraints. An* effect function *$\mathcal{E}$ w.r.t. Σ, $\mathcal{D}$, and $\mathcal{T}$ is a* partial *function $\mathcal{E} : \Sigma \times \mathcal{M}(\mathcal{T}) \to 2^{Lit}$. If $\mathcal{E}$ is defined for a pair $(\alpha, \mathcal{I}) \in \Sigma \times \mathcal{M}(\mathcal{T})$, then we say that α is* applicable *to $\mathcal{I}$. Otherwise, α is* not applicable *to $\mathcal{I}$.*

For the action formalism in [1], $\mathcal{E}(\alpha, \mathcal{I})$ consists of the literals of the conditional post-conditions whose condition is satisfied by $\mathcal{I}$. For the action formalism in [10], in addition to such direct effects, the set $\mathcal{E}(\alpha, \mathcal{I})$ also contains indirect effects generated by the causal relationships.

For every $\alpha \in \Sigma$, the effect function induces a binary relation $\Longrightarrow_{\alpha}^{\mathcal{E}}$ on $\mathcal{M}(\mathcal{T})$:

Definition 2. *Let $\mathcal{E}$ be an effect function w.r.t. Σ, $\mathcal{D}$, and $\mathcal{T}$. Then $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$ if the following conditions are satisfied:*

1. *α is applicable to $\mathcal{I}$,*
2. *there exists no $L \in Lit$ such that $\{L, \neg L\} \subseteq \mathcal{E}(\alpha, \mathcal{I})$,*
3. *$\Delta^{\mathcal{I}} = \Delta^{\mathcal{I}'}$ and $a^{\mathcal{I}} = a^{\mathcal{I}'}$ for all $a \in N_I$,*
4. *$A^{\mathcal{I}'} := (A^{\mathcal{I}} \cup \{a^{\mathcal{I}} \mid A(a) \in \mathcal{E}(\alpha, \mathcal{I})\}) \setminus \{a^{\mathcal{I}} \mid \neg A(a) \in \mathcal{E}(\alpha, \mathcal{I})\}$ for all $A \in N_C$,*
5. *$r^{\mathcal{I}'} := (r^{\mathcal{I}} \cup \{(a^{\mathcal{I}}, b^{\mathcal{I}}) \mid r(a, b) \in \mathcal{E}(\alpha, \mathcal{I})\}) \setminus \{(a^{\mathcal{I}}, b^{\mathcal{I}}) \mid \neg r(a, b) \in \mathcal{E}(\alpha, \mathcal{I})\}$ for all $r \in N_R$.*

If $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$ then we say: α transforms *the model $\mathcal{I}$ into the model $\mathcal{I}'$.*

Note that, for a given action $\alpha \in \Sigma$ and a model $\mathcal{I} \in \mathcal{M}(\mathcal{T})$, there is at most one $\mathcal{I}'$ such that $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$, i.e., the actions that we consider are *deterministic*.[2] There are several possible reasons why such an $\mathcal{I}'$ may not exist at all. First, the action may not be applicable to $\mathcal{I}$. In the formalism of [1], this corresponds to the fact that $\mathcal{I}$ does not satisfy the preconditions of the action. Second, the action may be applicable, but Condition 2 may not be satisfied. In [1], the action is then called inconsistent w.r.t. $\mathcal{I}$. Finally, it may be the case that Conditions 1 and 2 are satisfied, but the interpretation $\mathcal{I}'$ defined by the last three conditions is not a model of $\mathcal{T}$. In [10], actions for which this case occurs are also considered to be inconsistent.

Definition 3. *An action α is called* consistent *if, for every $\mathcal{I} \in \mathcal{M}(\mathcal{T})$ to which α is applicable, there is an $\mathcal{I}' \in \mathcal{M}(\mathcal{T})$ such that $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$.*

In the following, we will only consider action theories for which all actions are consistent.

[2] However, the complex actions that we will build from these deterministic atomic actions in Section 3 can well be non-deterministic due to the non-deterministic nature of the programming constructs used in Golog programs.

3 Golog Programs over DL Actions

In this section we define the syntax and semantics of a fragment of the action programming language Golog [8,11] that uses DL-based action theories as introduced in the previous section. Golog program expressions describe how a complex action is constructed from atomic actions using programming constructs and tests.

Definition 4. *Let Σ be a set of action names, $\alpha \in \Sigma$ and ψ a* test *that is built according to the following grammar:*

$$\psi ::= \varphi \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2$$

where φ is an assertion. A program expression *is defined inductively as follows.*

- *A* single action *$\alpha \in \Sigma$ is a program expression.*
- *The* empty program *$\langle\rangle$ is a program expression.*
- *If δ is a program expression, then the* non-deterministic iteration *of δ, denoted by $(\delta)^*$, is a program expression.*
- *If δ_1 and δ_2 are program expressions, then the* sequence *of δ_1 and δ_2, denoted by $(\delta_1; \delta_2)$, is a program expression.*
- *If ψ is a* test *and δ a program expression, then $\psi?; \delta$ is a program expression.*
- *If δ_1 and δ_2 are program expressions, then the* non-deterministic choice *between δ_1 and δ_2, denoted by $(\delta_1|\delta_2)$, is a program expression.*
- *If δ_1 and δ_2 are program expressions, then the* interleaving *of δ_1 and δ_2, denoted by $(\delta_1\|\delta_2)$, is a program expression.*

A DL-Golog program *$\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$ consists of a DL-based action theory $(\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E})$ and a program expression δ such that every action occurring in δ belongs to Σ and every assertion occurring in a test in δ belongs to $\mathcal{D}$.*

The *length* of program expressions, denoted by $|\delta|$, is defined to be the number of symbols (more precisely, the number of action names, tests, and constructors $*$, ;, $|$, $\|$) occurring in δ.

Note that the tests in the program expression can be used to model the control flow of a program whereas preconditions of actions (which in our setting are realized implicitly by the domain of the function $\mathcal{E}$) can be used to ensure that a single action is only applicable if it leads to a successor model. Together with the other constructs, tests can for example be used to express while-loops and if-then-else statements:

$$\textbf{while } \psi \textbf{ do } \delta \textbf{ endWhile} := (\psi?; \delta)^*; (\neg\psi?; \langle\rangle)$$
$$\textbf{if } \psi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} := \psi?; \delta_1 \mid \neg\psi?; \delta_2$$

Our notion of tests differs from the one in [8] in that tests are not viewed to be actions, and thus a test $\psi?$ alone is not a valid program expressions. In fact, viewing tests as actions creates problems when combined with interleaving since it may happen that a test is successfully executed, but then is followed by an interleaved action from another part of the program, which may in turn destroy the condition checked by the test. For example, consider the program expression obtained by interleaving of the two if-then-else statements **if** $A(a)$ **then** $\alpha_{-B(a)}$ **else** $\langle\rangle$ **endIf** and **if** $B(a)$ **then** $\alpha_{-A(a)}$ **else** $\langle\rangle$ **endIf**:

$$(A(a)?; \alpha_{-B(a)} \mid \neg A(a)?; \langle\rangle) \parallel (B(a)?; \alpha_{-A(a)} \mid \neg B(a)?; \langle\rangle), \tag{1}$$

where $\alpha_{-B(a)}$ and $\alpha_{-A(a)}$ are actions such that, for all models $\mathcal{I}$, $\mathcal{E}(\alpha_{-A(a)}, \mathcal{I}) = \{\neg A(a)\}$ and $\mathcal{E}(\alpha_{-B(a)}, \mathcal{I}) = \{\neg B(a)\}$, respectively. Let us also assume that the TBox describing the domain constraints is empty. Starting with a model $\mathcal{I}$ in which a belongs both to A and B, the program (1) should have two possible outcomes, depending on which if-then-else statement is executed first: either a belongs to A and not to B or a belongs to B and not to A. However, if tests are viewed as actions, then one could first successfully execute the test $A(a)?$, then the whole second if-then-else statement, and then the action $\alpha_{-B(a)}$, resulting in the unintended model in which a is contained neither in A nor in B. To avoid such unintended interactions between interleaving and tests, we consider a sequence of tests followed by an action as a unit, called guarded action, which must be executed in one atomic step.

Definition 5. *A* guarded action *is a program expression of the form*

$$\psi_1?; \ldots; \psi_n?; \alpha,$$

where $n \geq 0$, ψ_i for $i = 1, \ldots, n$ are tests, and $\alpha \in \Sigma$.

We will often use the symbol $\mathfrak{a}$ to denote a guarded action.[3].

In order to deal with terminating and non-terminating programs in a uniform way, we introduce an auxiliary action name ϵ, a fresh individual name p, and a fresh concept name *Term* to indicate that the program has terminated. We assume that these symbols do not occur in the input program, with the only exception that $\neg \mathit{Term}(p)$ belongs to the initial ABox. The effect of the action ϵ is predefined such that ϵ is applicable to all models of the domain constraints $\mathcal{T}$, and $\mathcal{E}(\epsilon, \mathcal{I}) = \{\mathit{Term}(p)\}$ holds for all models $\mathcal{I}$ of $\mathcal{T}$.

To define the semantics of DL-Golog programs, we introduce the functions $\mathsf{head}(\cdot)$ and $\mathsf{tail}(\cdot, \cdot)$. Intuitively, $\mathsf{head}(\delta)$ contains those guarded actions that can be executed first when executing the program expression δ. For $\mathfrak{a} \in \mathsf{head}(\delta)$, $\mathsf{tail}(\mathfrak{a}, \delta,)$ yields the remainder of the program, i.e., the part that still needs to be executed after $\mathfrak{a}$ has been executed. Due to the non-deterministic nature of Golog programs, $\mathsf{tail}(\mathfrak{a}, \delta,)$ is also a set of program expressions rather than a single one.

Definition 6. *The function $\mathsf{head}(\cdot)$ maps a program expression to a set of guarded actions. It is defined by induction on the structure of program expressions:*

$$\mathsf{head}(\langle\rangle) := \{\epsilon\};$$

$$\mathsf{head}(\alpha) := \{\alpha\} \text{ for all } \alpha \in \Sigma;$$

$$\mathsf{head}(\psi?; \delta) := \{\psi?; \mathfrak{a} \mid \mathfrak{a} \in \mathsf{head}(\delta)\};$$

[3] If $n = 0$, then the guarded action is actually an ordinary action, and thus $\mathfrak{a}$ may also denote an ordinary action.

$$
\begin{aligned}
\mathit{head}(\delta^*) &:= \{\epsilon\} \cup \mathit{head}(\delta);\\
\mathit{head}(\delta_1;\delta_2) &:= \{\mathfrak{a} \mid \mathfrak{a} = \psi_1?;...;\psi_n?;\alpha \in \mathit{head}(\delta_1) \wedge \alpha \neq \epsilon\} \cup\\
&\qquad \{\psi_1?;\ldots;\psi_n?;\psi_1'?;...;\psi_m'?;\alpha \mid \psi_1?;...;\psi_n?;\epsilon \in \mathit{head}(\delta_1) \wedge\\
&\qquad\qquad \psi_1'?;\ldots;\psi_m'?;\alpha \in \mathit{head}(\delta_2)\};\\
\mathit{head}(\delta_1|\delta_2) &:= \mathit{head}(\delta_1) \cup \mathit{head}(\delta_2);\\
\mathit{head}(\delta_1\|\delta_2) &:= \{\mathfrak{a} \mid \mathfrak{a} = \psi_1?;\ldots;\psi_n?;\alpha \in \mathit{head}(\delta_i) \wedge i \in \{1,2\} \wedge \alpha \neq \epsilon\} \cup\\
&\qquad \{\psi_1?;\ldots;\psi_n?;\psi_1'?;\ldots;\psi_m'?;\alpha \mid \psi_1?;\ldots;\psi_n?;\epsilon \in \mathit{head}(\delta_i) \wedge\\
&\qquad\qquad \psi_1'?;\ldots;\psi_m'?;\alpha \in \mathit{head}(\delta_j) \wedge\\
&\qquad\qquad \{1,2\} = \{i,j\}\}.
\end{aligned}
$$

Consider the definition of $\mathsf{head}(\delta_1;\delta_2)$. In this case, we first have to execute the program δ_1. Therefore, the first guarded action to be executed for the sequence is one of the heads of δ_1. However, if $\psi_1?;\ldots;\psi_n?;\epsilon$ is contained in the head of δ_1, then δ_1 can terminate successfully if the tests are satisfied. But in this case the subsequent program δ_2 still needs to be executed. Therefore, we must continue with a head of δ_2. This is achieved by replacing ϵ in $\psi_1?;\ldots;\psi_n?;\epsilon$ with a head of δ_2. Our definition of $\mathsf{head}(\delta_1\|\delta_2)$ can be explained in a similar way, and the other cases should be obvious.

Example 7. As an example, consider the following program expression

$$\delta = (\psi?;\alpha)^*;\big((\neg\psi)?;\langle\rangle\big),$$

which corresponds to a while-loop with condition ψ and body α. It is easy to see that $\mathsf{head}((\psi?;\alpha)^*) = \{\psi?;\alpha,\epsilon\}$. This means that, if we want to execute $(\psi?;\alpha)^*$, then in the first step we can execute α if ψ is satisfied, or we can terminate (indicated by the action ϵ). For the subsequent expression in δ we obtain $\mathsf{head}((\neg\psi)?;\langle\rangle) = \{(\neg\psi)?;\epsilon\}$. To determine $\mathsf{head}(\delta)$, we apply the definition of $\mathsf{head}(\delta_1;\delta_2)$, which yields $\mathsf{head}(\delta) = \{\psi?;\alpha,(\neg\psi)?;\epsilon\}$.

Thus, when starting to execute the while-loop, we can either execute α if ψ is satisfied, or terminate if ψ is not satisfied. Now consider the program expression ρ, which describes the interleaving of two while-loops:

$$\rho = \big((\psi_0?;\alpha_0)^*;(\neg\psi_0?;\langle\rangle)\big) \parallel \big((\psi_1?;\alpha_1)^*;(\neg\psi_1?;\langle\rangle)\big)$$

We have

$$\mathsf{head}(\rho) = \{\psi_0?;\alpha_0,\ \psi_1?;\alpha_1,\ \neg\psi_0?;\psi_1?;\alpha_1,\ \neg\psi_1?;\psi_0?;\alpha_0,\ \neg\psi_0?;\neg\psi_1?;\epsilon\}.$$

First, we have the choice to execute α_0 if ψ_0 is satisfied or α_1 if ψ_1 is satisfied. Furthermore, if ψ_0 is not satisfied and ψ_1 is satisfied, then it is possible to terminate the first while-loop since $\neg\psi_0;\epsilon \in \mathsf{head}((\psi_0?;\alpha_0)^*;(\neg\psi_0?;\langle\rangle))$. But we then have to consider the parallel while-loop. Since ψ_1 is satisfied we cannot terminate the whole program, but must continue with the second while-loop. This case is reflected by $\neg\psi_0?;\psi_1?;\alpha_1 \in \mathsf{head}(\rho)$. In case neither ψ_0 nor ψ_1 is satisfied, the program terminates, which explains $\neg\psi_0?;\neg\psi_1?;\epsilon \in \mathsf{head}(\rho)$.

Next, we need to define the program(s) that remain to be executed once a guarded action from the head has been executed.

Definition 8. *The function* $\mathsf{tail}(\cdot,\cdot)$ *maps a guarded action and a program expression to a set of program expressions.*

- *If* $\mathfrak{a} \notin \mathsf{head}(\delta)$*, then* $\mathsf{tail}(\mathfrak{a},\delta) = \emptyset$*.*
- *If* $\mathfrak{a} \in \mathsf{head}(\delta)$ *and* $\mathfrak{a} = \psi_1?; ...; \psi_n?; \epsilon$*, then* $\mathsf{tail}(\mathfrak{a},\delta) = \{\langle\rangle\}$*.*
- *If* $\mathfrak{a} \in \mathsf{head}(\delta)$ *and* $\mathfrak{a} = \psi_1?; ...; \psi_n?; \alpha$ *for* $\alpha \in \Sigma \setminus \{\epsilon\}$*, then* $\mathsf{tail}(\mathfrak{a},\delta)$ *is defined by induction on the combined size of* $\mathfrak{a}$ *and* δ*:*

$$\begin{aligned}
&\mathsf{tail}(\mathfrak{a},\langle\rangle) := \{\langle\rangle\};\\
&\mathsf{tail}(\mathfrak{a},\beta) := \{\langle\rangle\} \textit{ for } \beta \in \Sigma;^4\\
&\mathsf{tail}(\mathfrak{a},\delta^*) := \{\delta'; (\delta)^* \mid \delta' \in \mathsf{tail}(\mathfrak{a},\delta)\};\\
&\mathsf{tail}(\mathfrak{a},\psi?;\delta) := \mathsf{tail}(\psi_2?; \ldots; \psi_n?; \alpha, \delta);^5\\
&\mathsf{tail}(\mathfrak{a},\delta_1;\delta_2) := \{\delta'; \delta_2 \mid \delta' \in \mathsf{tail}(\mathfrak{a},\delta_1)\} \cup\\
&\qquad \{\delta'' \mid \exists j, 0 \leq j \leq n \textit{ s.t. } \psi_1?; ...; \psi_j?; \epsilon \in \mathsf{head}(\delta_1) \wedge\\
&\qquad\quad \psi_{j+1}?; ...; \psi_n?; \alpha \in \mathsf{head}(\delta_2) \wedge\\
&\qquad\quad \delta'' \in \mathsf{tail}(\psi_{j+1}?; ...; \psi_n?; \alpha, \delta_2)\};\\
&\mathsf{tail}(\mathfrak{a},\delta_1|\delta_2) := \mathsf{tail}(\mathfrak{a},\delta_1) \cup \mathsf{tail}(\mathfrak{a},\delta_2);\\
&\mathsf{tail}(\mathfrak{a},\delta_1\|\delta_2) := \{\delta'\|\delta_2 \mid \delta' \in \mathsf{tail}(\mathfrak{a},\delta_1)\} \cup \{\delta_1\|\delta' \mid \delta' \in \mathsf{tail}(\mathfrak{a},\delta_2)\} \cup\\
&\qquad \{\delta'' \mid \exists j, 0 \leq j \leq n \textit{ s.t. } \psi_1?; ...; \psi_j?; \epsilon \in \mathsf{head}(\delta_i) \wedge\\
&\qquad\quad \psi_{j+1}?; ...; \psi_n?; \alpha \in \mathsf{head}(\delta_{i'}) \wedge\\
&\qquad\quad \delta'' \in \mathsf{tail}(\psi_{j+1}?; ...; \psi_n?; \alpha, \delta_{i'}) \wedge \{1,2\} = \{i,i'\}\}.
\end{aligned}$$

In the definitions of $\mathsf{tail}(\mathfrak{a},\delta^*)$*,* $\mathsf{tail}(\mathfrak{a},\delta_1;\delta_2)$*, and* $\mathsf{tail}(\mathfrak{a},\delta_1\|\delta_2)$*, we omit* δ' *if* $\delta' = \langle\rangle$*.*

An example illustrating the definition of the tail function can be found in [6].

Intuitively, executing a program δ means first executing a guarded action of its head, then a guarded action of the head of its tail, etc. We call a program expression that can be reached by a sequence of such head and tail applications a reachable subprogram.

Definition 9. *Let* δ *be a program expression. The program expression* ρ *is a* reachable subprogram *of* δ *if there is an* $n \geq 0$ *and program expressions* $\delta_0, \delta_1, ..., \delta_n$ *such that* $\delta_0 = \delta$*,* $\delta_n = \rho$*, and for all* $i = 0, \ldots, n-1$ *there exists* $\mathfrak{a}_i \in \mathsf{head}(\delta_i)$ *such that* $\delta_{i+1} \in \mathsf{tail}(\mathfrak{a}_i, \delta_i)$*. We denote the set of all reachable subprograms of* δ *by* $\mathsf{sub}(\delta)$*.*

The following lemma is vital for our proof of decidability of the verification problem since it shows that there are only finitely many reachable subprograms of a given program expression.

[4] Note that $\mathfrak{a} \in \mathsf{head}(\beta)$ implies $\mathfrak{a} = \beta$.

[5] Note that $\mathfrak{a} = \psi_1?; ...; \psi_n?; \alpha \in \mathsf{head}(\psi?;\delta)$ implies $\psi = \psi_1$.

Lemma 10. *Let δ be a program expression.*

1. *The cardinality of $\mathsf{sub}(\delta)$ is exponentially bounded in the size $|\delta|$ of δ.*
2. *If δ does not contain the interleaving constructor, then the size of $\mathsf{sub}(\delta)$ is polynomially bounded in the size $|\delta|$ of δ.*

A proof of this lemma can be found in [6]. Note that, in the presence of the interleaving operator, the exponential bound is actually reached. In fact, consider the program expression $\delta = \alpha_1 \parallel (\alpha_2 \parallel \ldots (\alpha_{n-1} \parallel \alpha_n)\ldots)$. We claim that $\mathsf{sub}(\delta)$ contains at least 2^n many reachable subprograms. In fact, it is easy to see that for every subset $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ with $i_1 \leq \ldots \leq i_k$ the expression $\alpha_{i_1} \parallel (\alpha_{i_2} \parallel \ldots (\alpha_{i_{k-1}} \parallel \alpha_{i_k})\ldots)$ is a reachable subprogram of δ.

Now we are ready to define the semantics of a DL-Golog program. Similar to the semantics introduced in [7], a program induces an infinite transition system. The states of this transition system are *program configurations*, which are pairs consisting of a program expression and a model of the TBox. To make a transition from one configuration to another, we pick an applicable guarded action from the head of the program expression, and then transform the model and the program expression using the semantics of actions and the tail function.

Definition 11. *Let $\mathfrak{a} = \psi_1?; \ldots; \psi_n?; \alpha$ be a guarded action and $\mathcal{I}$ an interpretation. We say that $\mathfrak{a}$ is* applicable *to $\mathcal{I}$ iff $\mathcal{I} \models \psi_i$ holds for all $i \in \{1, \ldots, n\}$ and the action α is applicable to $\mathcal{I}$.*[6]

A program configuration of the form $(\mathcal{I}, \delta)$ is called a *failing configuration* if no $\mathfrak{a} \in \mathsf{head}(\delta)$ is applicable to $\mathcal{I}$. To indicate such a crash of the program execution, we add another predefined action $\mathfrak{f}$ to Σ. The action $\mathfrak{f}$ is applicable to all models of $\mathcal{T}$, and we set $\mathcal{E}(\mathfrak{f}, \mathcal{I}) := \{\mathit{Fail}(p)\}$ where *Fail* is a fresh concept name. In addition we require that $\neg\mathit{Fail}(p) \in \mathcal{A}$ for the initial ABox $\mathcal{A}$.

Definition 12 (program semantics). *Let $\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$ be a DL-Golog program. The* transition system *$T_{\mathcal{P}} = (Q, \rightarrow, I)$ induced by $\mathcal{P}$ consists of the set of* program configurations *$Q := \mathcal{M}(\mathcal{T}) \times \mathsf{sub}(\delta)$ as well as a transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$ and a set of initial configurations $I \subseteq Q$, which are defined as follows:*

- *The* initial configurations *are defined as $I := \{(\mathcal{I}, \delta) \mid \mathcal{I} \in \mathcal{M}(\mathcal{A}) \cap \mathcal{M}(\mathcal{T})\}$.*
- *We have $\big((\mathcal{I}, \rho), \alpha, (\mathcal{I}', \rho')\big) \in \rightarrow$ (written as $(\mathcal{I}, \rho) \xrightarrow{\alpha} (\mathcal{I}', \rho')$) if one of the following conditions is satisfied:*
  1. *There exists $\mathfrak{a} = \psi_1?, \ldots, \psi_n?; \alpha \in \mathit{head}(\rho)$ such that $\mathfrak{a}$ is applicable to $\mathcal{I}$ and it holds that $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$ and $\rho' \in \mathit{tail}(\mathfrak{a}, \rho)$.*
  2. *There is no $\mathfrak{a} \in \mathit{head}(\rho)$ such that $\mathfrak{a}$ is applicable to $\mathcal{I}$ and it holds that $\alpha = \mathfrak{f}$, $\rho = \rho'$ and $\mathcal{I} \Longrightarrow_{\mathfrak{f}}^{\mathcal{E}} \mathcal{I}'$.*

[6] Recall that tests are Boolean combinations of assertions. The notion of satisfaction in an interpretation is extended in the obvious way from assertions to such Boolean combinations.

Since we assume that all actions are consistent, there always exists a successor configuration $\mathcal{I}'$ in Case 1 of the definition of $\xrightarrow{\alpha}$. This fact, together with the presence of the predefined actions ϵ and $\mathfrak{f}$, ensures that every configuration in Q has at least one successor configuration. In particular, every initial configuration $(\mathcal{I}_0, \delta_0) \in I$ is the starting point of an infinite path in the transition system $T_{\mathcal{P}}$:

$$\pi = (\mathcal{I}_0, \delta_0) \xrightarrow{\alpha_0} (\mathcal{I}_1, \delta_1) \xrightarrow{\alpha_1} (\mathcal{I}_2, \delta_2) \xrightarrow{\alpha_3} (\mathcal{I}_3, \delta_3) \xrightarrow{\alpha_4} \ldots$$

We call such a path a *run* of the DL-Golog program $\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$. The *action trace* of the run π is an infinite word $w(\pi)$ over Σ with $w(\pi) = \alpha_0\alpha_1\alpha_2\alpha_3\ldots$. The infinite sequence of interpretations $\mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3, \ldots$ occurring in the configurations in π is denoted by $\mathfrak{I}(\pi)$. By the definition of $\xrightarrow{\alpha}$ we have $\mathcal{I}_i \Longrightarrow^{\mathcal{E}}_{\alpha_i} \mathcal{I}_{i+1}$ for all $i \geq 0$.

The introduction of the predefined actions ϵ and $\mathfrak{f}$ allows us to treat non-terminating, terminating, and failing runs of a given program in a uniform way. Let $\Delta := \Sigma \setminus \{\epsilon, \mathfrak{f}\}$ denote the set of proper actions. A run π of a program $\mathcal{P}$ is called

- a *terminating run* if $w(\pi) \in \Delta^* \cdot \{\epsilon\}^\omega$,[7]
- a *failing run* if $w(\pi) \in \Delta^* \cdot \{\mathfrak{f}\}^\omega$,
- a *non-terminating run* if $w(\pi) \in \Delta^\omega$.

It is easy to show that, according to this definition, every run of a program is either terminating, non-terminating, or failing.

4 Verifying Temporal Properties of DL-Golog Programs

First, we need to introduce the temporal logic used to specify properties of runs of a program. As in [5], we use a variant of the temporalized DL $\mathcal{ALC}$-LTL [3], which we call $\mathcal{ALCO}$-LTL since it is based on the more expressive DL $\mathcal{ALCO}$. The syntax is the same as for propositional linear time logic (LTL), but in place of propositions, assertions are used.[8] More precisely, $\mathcal{ALCO}$-LTL formulas are built according to the following grammar:

$$\Phi ::= \varphi \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \mathsf{X}\Phi \mid \Phi_1 \,\mathsf{U}\, \Phi_2$$

where φ stands for assertions. As usual, $\Diamond\Phi$ (*eventually*) and $\Box\Phi$ (*always*) are used as abbreviations for $\top(a) \,\mathsf{U}\, \Phi$ and $\neg\Diamond\neg\Phi$, respectively. Moreover, $\Phi_1 \rightarrow \Phi_2$ abbreviates $\neg\Phi_1 \vee \Phi_2$.

The semantics of $\mathcal{ALCO}$-LTL is based on the notion of an *$\mathcal{ALCO}$-LTL structure*, which is an infinite sequence of interpretations $\mathfrak{I} = (\mathcal{I}_i)_{i=0,1,2,\cdots}$ over a

[7] i.e., a finite sequence of proper actions followed by an infinite sequence using only the action ϵ.

[8] In $\mathcal{ALC}$-LTL, also GCIs can occur in place of propositions, but this is not needed in the context of this paper. The domain constraints constitute a global TBox that must hold at every time point.

common domain Δ such that $a^{\mathcal{I}_i} = a^{\mathcal{I}_j}$ is satisfied for all $a \in N_I$ and all $i, j \in \{0, 1, 2, \ldots\}$. Let Φ be an $\mathcal{ALCO}$-LTL formula, $\mathfrak{I}$ an $\mathcal{ALCO}$-LTL structure, and $i \in \{0, 1, 2, \ldots\}$ a time point. Validity of Φ in $\mathfrak{I}$ at time i, denoted by $\mathfrak{I}, i \models \Phi$, is defined as follows:

$$\begin{array}{ll}
\mathfrak{I}, i \models \varphi & \text{iff } \mathcal{I}_i \models \varphi, \\
\mathfrak{I}, i \models \neg\Phi & \text{iff } \mathfrak{I}, i \not\models \Phi, \\
\mathfrak{I}, i \models \Phi_1 \wedge \Phi_2 & \text{iff } \mathfrak{I}, i \models \Phi_1 \text{ and } \mathfrak{I}, i \models \Phi_2, \\
\mathfrak{I}, i \models \Phi_1 \vee \Phi_2 & \text{iff } \mathfrak{I}, i \models \Phi_1 \text{ or } \mathfrak{I}, i \models \Phi_2, \\
\mathfrak{I}, i \models \mathsf{X}\Phi & \text{iff } \mathfrak{I}, i+1 \models \Phi, \\
\mathfrak{I}, i \models \Phi_1 \,\mathsf{U}\, \Phi_2 & \text{iff } \exists k \geq i : \mathfrak{I}, k \models \Phi_2 \text{ and } \forall j, i \leq j < k : \mathfrak{I}, j \models \Phi_1
\end{array}$$

We can use $\mathcal{ALCO}$-LTL formulas to specify desired or unwanted properties of (runs of) DL-Golog programs.

Definition 13 (verification problem). *Let $\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$ be a DL-Golog program and Φ an $\mathcal{ALCO}$-LTL formula such that Φ contain only assertions from the set $\mathcal{D}$. The formula Φ is* valid *in $\mathcal{P}$, written as $\mathcal{P} \models \Phi$, if for all runs π of $\mathcal{P}$ it holds that $\mathfrak{I}(\pi), 0 \models \Phi$. The formula Φ is* satisfiable *in $\mathcal{P}$ if there exists a run π of $\mathcal{P}$ such that $\mathfrak{I}(\pi), 0 \models \Phi$.*

Using the literals $\mathit{Term}(p)$ and $\mathit{Fail}(p)$, we can encode variants of the verification problem into the formula. For example, we can check whether there is a failing run by testing satisfiability of the formula $\Diamond \mathit{Fail}(p)$. The fact that all runs of the program are terminating corresponds to the validity of the formula $\Diamond \mathit{Term}(p)$. In order to verify that all infinite runs of the program satisfy Φ, we can test validity of the formula $\Box(\neg \mathit{Fail}(p) \wedge \neg \mathit{Term}(p)) \rightarrow \Phi$.

Clearly, Φ is valid in $\mathcal{P}$ iff $\neg\Phi$ is unsatisfiable in $\mathcal{P}$. Therefore, it is sufficient to focus on showing decidability of the satisfiability problem. To obtain decidability, we need to impose additional restrictions on our action theories. Basically, we need to ensure that the effect function is computable and that we can construct a finite abstraction of the infinite transition system $T_{\mathcal{P}}$ that contains enough information on the runs of the program to enable verification based on this abstraction.

In order to obtain this finite abstraction, we use the notion of a static type to partition the infinite set of models of the TBox into finitely many equivalence classes of elements of the same type.

Definition 14. *Given a DL-based action theory $(\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E})$, a* static type *for this theory is a set $\mathfrak{t} \subseteq \mathcal{D}$ such that the ABox $\mathfrak{t}$ is consistent w.r.t. $\mathcal{T}$ and for all $\neg\varphi \in \mathcal{D}$ we have $\varphi \in \mathfrak{t}$ iff $\neg\varphi \notin \mathfrak{t}$. The* static type of a model $\mathcal{I}$ *of $\mathcal{T}$ is defined as $\mathsf{s\text{-}type}(\mathcal{I}) := \{\varphi \in \mathcal{D} \mid \mathcal{I} \models \varphi\}$.*

Obviously, given a model $\mathcal{I}$ of $\mathcal{T}$, the set $\mathsf{s\text{-}type}(\mathcal{I})$ is indeed a static type. Conversely, for every static type $\mathfrak{t}$, there is a model $\mathcal{I}$ of $\mathcal{T}$ such that $\mathfrak{t} = \mathsf{s\text{-}type}(\mathcal{I})$. Intuitively, models of the same static type cannot be distinguished by an assertion occurring in $\mathcal{D}$.

We are now ready to formulate conditions on DL-based action theories that ensure decidability of the satisfiability problem.

Definition 15 (admissibility). *The DL-based action theory $(\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E})$ is called* admissible *if all its actions are consistent and the conditions (A1), (A2), (A3) are satisfied for models $\mathcal{I}$ and $\mathcal{J}$ of $\mathcal{T}$:*

(A1) *If $\textsf{s-type}(\mathcal{I}) = \textsf{s-type}(\mathcal{J})$, then α is applicable to $\mathcal{I}$ iff α is applicable to $\mathcal{J}$*
(A2) *If $\textsf{s-type}(\mathcal{I}) = \textsf{s-type}(\mathcal{J})$ and α is applicable to $\mathcal{I}$, then $\mathcal{E}(\alpha, \mathcal{I}) = \mathcal{E}(\alpha, \mathcal{J})$.*

If (A1) and (A2) are satisfied and $\mathfrak{t}$ is a static type, then we define $\mathcal{E}(\alpha, \mathfrak{t}) := \mathcal{E}(\alpha, \mathcal{I})$, where $\mathcal{I}$ is an arbitrary model of $\mathcal{T}$ with $\mathfrak{t} = \textsf{s-type}(\mathcal{I})$.

(A3) *For a given static type $\mathfrak{t}$, it is decidable whether $\mathcal{E}(\alpha, \mathfrak{t})$ is defined. If it is defined, then this set can be effectively computed.*

It is easy to see that the action theories introduced in [1,4] are indeed admissible (if all actions are required to be consistent). For example, in both formalisms applicability of an action α to a model $\mathcal{I}$ depends on whether the preconditions of α are satisfied in $\mathcal{I}$. Since these preconditions are assertions in $\mathcal{D}$, (A1) is obviously satisfied.

Unfortunately, given two models of the same static type and an action that is applicable to these models, the models obtained by applying the action need not have the same static type. This is illustrated by the following example.

Example 16. Assume that $\mathcal{A} = \{B(b), r(a, b), \exists r.B(a)\}$,

$$\mathcal{D} = \{B(b), \neg B(b), r(a, b), \neg r(a, b), \exists r.B(a), \neg\exists r.B(a)\},$$

and $\mathcal{T} = \emptyset$, and consider an action α with the effect function

$$\mathcal{E}(\alpha, \mathcal{I}) = \{\neg B(b)\} \text{ if } \mathcal{I} \models \exists r.B(a).$$

Otherwise, $\mathcal{E}(\alpha, \mathcal{I})$ is undefined. Intuitively, $\exists r.B(a)$ is the precondition of α and $\neg B(b)$ the effect. It is easy to see that this action theory is admissible.

Consider two models $\mathcal{I}_1$ and $\mathcal{I}_2$ of $\mathcal{A}$ over the same domain with

$$\Delta^{\mathcal{I}_1} = \Delta^{\mathcal{I}_2} = \{a, b, d\}, r^{\mathcal{I}_1} = \{(a, b)\}, r^{\mathcal{I}_2} = \{(a, b), (a, d)\}, B^{\mathcal{I}_1} = B^{\mathcal{I}_2} = \{b, d\},$$

such that $a^{\mathcal{I}_i} = a$, $b^{\mathcal{I}_i} = b$ $(i = 1, 2)$ and $d \notin \{a, b\}$. Clearly, we have $\textsf{s-type}(\mathcal{I}_1) = \textsf{s-type}(\mathcal{I}_2)$ and $\mathcal{E}(\alpha, \mathcal{I}_1) = \mathcal{E}(\alpha, \mathcal{I}_2) = \{\neg B(b)\}$. However, for the interpretations $\mathcal{I}_1', \mathcal{I}_2'$ with $\mathcal{I}_1 \Longrightarrow_{\alpha}^{\{\neg B(b)\}} \mathcal{I}_1'$ and $\mathcal{I}_2 \Longrightarrow_{\alpha}^{\{\neg B(b)\}} \mathcal{I}_2'$, it holds that $\mathcal{I}_1' \not\models \exists r.B(a)$ and $\mathcal{I}_2' \models \exists r.B(a)$. Therefore, $\mathcal{I}_1'$ and $\mathcal{I}_2'$ do not have the same static type. Also note that α is applicable to $\mathcal{I}_2'$, but not to $\mathcal{I}_1'$.

This example shows that in general the mapping of models to their static types does *not* preserve the transition relation $\Longrightarrow_{\alpha}^{\mathcal{E}}$ on models for an action α. Therefore, the transition relation cannot be lifted to a transition relation on static types. To overcome this problem, we define an extended notion of types, called dynamic types. This requires the introduction of some more notation. Let $\mathcal{I}$ be an interpretation and $E \subseteq \mathit{Lit}$ a *non-contradictory* set of literals i.e., a set such that there is no L such that $\{L, \neg L\} \subseteq E$. The *updated interpretation* $\mathcal{I}^E$ w.r.t. E is defined as follows:

- $A^{\mathcal{I}^E} := (A^{\mathcal{I}} \cup \{a^{\mathcal{I}} \mid A(a) \in E\}) \setminus \{a^{\mathcal{I}} \mid \neg A(a) \in E\}$ for all $A \in N_C$ and
- $r^{\mathcal{I}^E} := (r^{\mathcal{I}} \cup \{(a^{\mathcal{I}}, b^{\mathcal{I}}) \mid r(a,b) \in E\}) \setminus \{(a^{\mathcal{I}}, b^{\mathcal{I}}) \mid \neg r(a,b) \in E\}$ for all $r \in N_R$.

Note that $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$ implies $\mathcal{I}' = \mathcal{I}^{\mathcal{E}(\alpha,\mathcal{I})}$. Using the notation $\neg E := \{\neg L \mid L \in E\}$ (modulo elimination of double negation), we can reduce iterated update operations to a single one: For two non-contradictory sets of literals E and E' we have

$$(\mathcal{I}^E)^{E'} = \mathcal{I}^{((E \setminus \neg E') \cup E')}. \tag{2}$$

Given a DL-Golog program $\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$, we call a pair (φ, E) consisting of an assertion $\varphi \in \mathcal{D}$ and a non-contradictory set of literals $E \subseteq \mathit{Lit}$ a *type element* for $\mathcal{P}$. The set of all type elements for $\mathcal{P}$ is denoted by $\mathit{TE}(\mathcal{P})$.

Definition 17 (dynamic types). *Let $\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$ be a DL-Golog program and $\mathcal{I}$ a model of $\mathcal{T}$. The* dynamic type of $\mathcal{I}$ *is defined as*

$$\textit{d-type}(\mathcal{I}) := \{(\varphi, E) \in \mathit{TE}(\mathcal{P}) \mid \mathcal{I}^E \models \varphi\}.$$

A set $\mathfrak{t} \subseteq \mathit{TE}(\mathcal{P})$ is called a dynamic type *if there is an interpretation $\mathcal{I}$ such that $\mathfrak{t} = \textit{d-type}(\mathcal{I})$.*

Since $(\varphi, \emptyset) \in \mathsf{d\text{-}type}(\mathcal{I})$ iff $\mathcal{I} \models \varphi$ iff $\varphi \in \mathsf{s\text{-}type}(\mathcal{I})$, models with the same dynamic type also have the same static type. Given a dynamic type $\mathfrak{t}$, we denote the corresponding static type by $\mathcal{A}(\mathfrak{t})$, i.e., $\mathcal{A}(\mathfrak{t}) := \{\varphi \mid (\varphi, \emptyset) \in \mathfrak{t}\}$. The next lemma shows that dynamic types have the desired property that the execution of an action transforms models of the same type again into models of the same type. The dynamic type of the successor models can easily be computed using the identity (2).

Lemma 18. *Let $\mathcal{I}, \mathcal{J}$ be models of $\mathcal{T}$ and α an action such that $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$ and $\mathcal{J} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{J}'$.*

1. *If $\textit{d-type}(\mathcal{I}) = \textit{d-type}(\mathcal{J})$, then $\textit{d-type}(\mathcal{I}') = \textit{d-type}(\mathcal{J}')$.*
2. *If $\mathcal{E}(\alpha, \mathcal{I}) = E'$ and $(\varphi, E) \in \mathit{TE}(\mathcal{P})$, then $(\varphi, E) \in \textit{d-type}(\mathcal{I}')$ iff $(\varphi, (E' \setminus \neg E) \cup E) \in \textit{d-type}(\mathcal{I})$.*

Given a subset $\mathfrak{t}$ of $\mathcal{D}$, DL reasoning can obviously be used to decide whether $\mathfrak{t}$ is a static type or not. Given a subset $\mathfrak{t}$ of $\mathit{TE}(\mathcal{P})$, it is also possible to reduce the check whether it is a dynamic type to DL reasoning, but how to do this is less obvious. In [6] we show how to adapt the *reduction approach* developed in [1] for deciding the projection problem to this purpose. The idea is to encode the model $\mathcal{I}$ and the updated interpretations $\mathcal{I}^E$ into one single model of an appropriate TBox and ABox. Basically, for every non-contradictory set of literals E we introduce renamed copies of all concept and role names, which also yield renamed copies $\varphi^{(E)}$ of all assertion $\varphi \in \mathcal{D}$. The reduction TBox $\mathcal{T}_{\text{red}}$ and the reduction ABox $\mathcal{A}_{\text{red}}(\mathfrak{t})$ are then constructed such that the following lemma holds.

Lemma 19. *Let $\mathfrak{t} \subseteq TE(\mathcal{P})$. Then $\mathfrak{t}$ is a dynamic type iff the following two properties are satisfied:*

1. *For all $(\neg\varphi, E) \in TE(\mathcal{P})$ it holds that $(\varphi, E) \in \mathfrak{t}$ iff $(\neg\varphi, E) \notin \mathfrak{t}$.*
2. *There exists a model $\mathcal{J}$ of $\mathcal{A}_{red}(\mathfrak{t})$ and $\mathcal{T}_{red}$ such that $\mathcal{J} \models \varphi^{(E)}$ holds for all $(\varphi, E) \in \mathfrak{t}$.*

Basically, our the decision procedure for the satisfiability of an $\mathcal{ALCO}$-LTL formula Φ in a DL-Golog program $\mathcal{P}$ works as follows. Instead of the transition system $T_{\mathcal{P}}$ we consider the quotient system $\widehat{T}_{\mathcal{P}}$ that is obtained from $T_{\mathcal{P}}$ by replacing models by their dynamic types. Since there are only finitely many dynamic types and reachable subprograms, this quotient transition system is finite. Similar to the well-known construction for propositional LTL, the formula Φ can be translated into a Büchi automaton $\mathcal{B}_{\Phi}$ such that Φ is satisfiable iff $\mathcal{B}_{\Phi}$ accepts a non-empty language. In order to test satisfiability in $\mathcal{P}$, we consider the Büchi automaton obtained by building the product of $\mathcal{B}_{\Phi}$ and $\widehat{T}_{\mathcal{P}}$, and test it for non-emptiness. In principle, this test boils down to a reachability problem (is there a final state that can be reached from an initial state and from itself). To solve this problem, we guess a dynamic type that satisfies the initial ABox (using the decision procedure for dynamic types obtained from Lemma 18) and pair it with an appropriate initial state of the Büchi automaton. Then we make transitions in the product automaton, where the transitions in the component corresponding to $\widehat{T}_{\mathcal{P}}$ are realized using the head and tail function for computing the new program expression and 2. of Lemma 18 for computing the new dynamic type. More details on how this decision procedure works can be found in [6].

Theorem 20. *For DL-Golog programs $\mathcal{P}$ whose underlying action theory is admissible, satisfiability of an $\mathcal{ALCO}$-LTL formula in $\mathcal{P}$ is decidable.*

The exact complexity of this decision procedure depends, of course, also on the complexity of computing the effect function $\mathcal{E}$. For the action formalisms in [1,4], this is the same complexity as reasoning in $\mathcal{ALCO}$ (i.e., ExpTime if the TBox contains GCIs, and PSpace if the TBox is empty or acyclic). In this case, the overall complexity of deciding satisfiability in a DL-Golog program is majorized by the complexity of testing whether a set $\mathfrak{t} \subseteq TE(\mathcal{P})$ is a dynamic type or not. Due to the fact that there are exponentially many type elements, the reduction TBox and ABox are of exponential size, and thus the complexity of this test is 2ExpTime if the TBox contains GCIs, and ExpSpace if the TBox is empty or acyclic.

5 Conclusion

We have shown first results on how to verify temporal properties of Golog programs that use Description Logic actions. The two main contributions of this paper are the following. First, we have defined a semantics for DL-Golog programs that is similar to the semantics of Golog program introduced in [7], but treats non-terminating, terminating, and failing runs of a given program in a uniform way. Second, we have introduced the new notion of a dynamic type, and have shown that it allows us to obtain a finite, semantics-preserving abstraction of the infinite transition system induced by a program. This is the main reason why the verification problem turns out to be decidable.

In our future work, we intend to extend our results both to more expressive action theories (e.g. ones with non-deterministic atomic actions) and to additional program construct. Regarding the temporal logic used to specify properties, we will also look at CTL and CTL* in place of LTL.

References

1. Baader, F., Lutz, C., Milicic, M., Sattler, U., Wolter, F.: Integrating description logics and action formalisms: First results. In: Proc. AAAI 2005 (2005)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
3. Baader, F., Ghilardi, S., Lutz, C.: LTL over description logic axioms. ACM Trans. Comput. Log. 13(3) (2012)
4. Baader, F., Lippmann, M., Liu, H.: Using causal relationships to deal with the ramification problem in action formalisms based on description logics. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 82–96. Springer, Heidelberg (2010)
5. Baader, F., Liu, H., ul Mehdi, A.: Verifying properties of infinite sequences of description logic actions. In: Proc. ECAI 2010 (2010)
6. Baader, F., Zarrieß, B.: Verification of golog programs over description logic actions. LTCS-Report 13-08, Chair of Automata Theory, TU Dresden, Dresden, Germany (2013), `http://lat.inf.tu-dresden.de/research/reports.html`
7. Claßen, J., Lakemeyer, G.: A logic for non-terminating golog programs. In: Proc. KR 2008 (2008)
8. De Giacomo, G., Lespérance, Y., Levesque, H.J.: Congolog, a concurrent programming language based on the situation calculus. Artif. Intell. 121(1-2), 109–169 (2000)
9. De Giacomo, G., Lespérance, Y., Patrizi, F.: Bounded situation calculus action theories and decidable verification. In: Proc. KR 2012 (2012)
10. Liu, H., Lutz, C., Miličić, M., Wolter, F.: Reasoning about actions using description logics with general TBoxes. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 266–279. Springer, Heidelberg (2006)
11. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: Golog: A logic programming language for dynamic domains. J. Log. Program. 31(1-3), 59–83 (1997)
12. Pnueli, A.: The temporal logic of programs. In: Proc. FOCS 1977 (1977)

Specification and Verification of Linear Dynamical Systems: Advances and Challenges

Joël Ouaknine

Department of Computer Science, Oxford University, UK
joel@cs.ox.ac.uk

Abstract. *Dynamical systems* are mathematical models in which the state of a system at any point in time is represented by a vector of variables, with a fixed rule determining the evolution of these variables over time. *Continuous linear dynamical systems* are governed by a multivariate linear differential equation, whereas *discrete-time linear dynamical systems* are governed by a linear transformation. In both cases, given initial values for the variables, the rule uniquely determines the evolution of the system over time.

Particular instances of such systems have been studied for decades in various areas of science and engineering, often either through simulations or in terms of long-run behaviour: existence and uniqueness of attractors, fixed points, or periodic points, sensitivity to initial conditions, etc. From the point of view of computer science, it is somewhat surprising to note the relative scarcity of literature on *decision problems* concerning linear dynamical systems, e.g., whether a fixed point or a particular region will actually be reached in finite time, whether a variable will assume negative values infinitely often, etc. Such questions, in turn, have numerous applications in a wide array of scientific areas, such as theoretical biology (analysis of L-systems, population dynamics), microeconomics (stability of supply-and-demand equilibria in cyclical markets), software verification (termination of linear programs), probabilistic model checking (reachability and approximation in Markov chains, stochastic logics), quantum computing (threshold problems for quantum automata), as well as combinatorics, formal languages, statistical physics, etc.

In this talk, I will describe recent advances in decision problems for linear dynamical systems, and discuss various open problems, both algorithmic in nature and in terms of defining suitable—i.e., expressive as well as tractable, or at least decidable—formalisms and frameworks for formulating requirements on linear dynamical systems.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, p. 197, 2013.

Obtaining Finite Local Theory Axiomatizations via Saturation

Matthias Horbach and Viorica Sofronie-Stokkermans

University Koblenz-Landau and Max-Planck-Institut für Informatik Saarbrücken

Abstract. In this paper we present a method for obtaining local sets of clauses from possibly non-local ones. For this, we follow the work of Basin and Ganzinger and use saturation under a version of ordered resolution. In order to address the fact that saturation can generate infinite sets of clauses, we use constrained clauses and show that a link can be established between saturation and locality also for constrained clauses: This often allows us to give a finite representation of possibly infinite saturated sets of clauses.

1 Introduction

Many problems in mathematics and computer science can be reduced to proving the satisfiability of conjunctions of literals in a background theory. It is therefore very important to identify situations where reasoning in extensions and combinations of theories can be done efficiently and accurately. The most important issues which need to be addressed in this context are:

(i) finding possibilities of reducing the search space without losing completeness,
(ii) making modular or hierarchical reasoning possible.

In [18], we introduced a class of theory extensions (which we named local) for which both aspects above can be addressed: (i) complete instantiation schemes exist, and (ii) hierarchical and modular reasoning is possible. However, locality is a property of an axiomatization of a theory rather than of the theory itself. Therefore, it is very important to *recognize locality* of a set of clauses, and *to obtain local axiomatizations* – for instance by transforming non-local sets of clauses into local ones. In [4, 5], Basin and Ganzinger presented a link between (order)-locality and saturation under ordered (hyper)resolution. Their result allows to obtain, by saturation, local axiomatizations for a theory from non-local ones. The drawback is that the size of the saturated sets of clauses is often very large and sometimes the saturation process may not terminate. The main contributions of this paper are:

- In order to obtain finite representations of possibly infinite sets of clauses we use *constrained clauses*.
- We use a sound and complete ordered resolution and superposition calculus for constrained clauses. In cases when a classical saturation process might not terminate, the use of constrained clauses allows us to give a finite representation for possibly infinite sets of clauses.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 198–213, 2013.

- We show that for certain types of constrained clauses a link can be established between saturation in our calculus and order locality.
- We indicate the limitations of our approach.

In [13] we identified situations in which the combination of two local theory extensions of a base theory $\mathcal{T}_0$ is a local extension of $\mathcal{T}_0$. The assumptions on the component theories are syntactic, so can be easily checked. Together with the results presented in this paper, this allows to prove the locality of combinations of theories, or to obtain local axiomatizations for combinations of theories.

We briefly discuss the relationships between our results and existing work.

Ordered resolution and superposition are often used to devise decision procedures for various theories, cf. e.g. [2, 14, 1]. Our approach allows us to consider, in addition, situations in which the saturated sets are not finite, by giving finite representations for them.

The connection between saturation under ordered resolution and (order) locality was studied by Basin and Ganzinger in [4, 5]. It is however relatively easy to see that no link between saturation under superposition and (order) locality can be established in general. Therefore, if we start with a set N of clauses in first-order logic with equality, then in order to use the results in [4, 5] for proving the locality of N (or for constructing an equivalent local axiomatization) we usually need to saturate $N \cup EQ$ under ordered resolution (where EQ is the set of congruence axioms). The results presented in this paper allow us to identify a class of clauses in first-order logic with equality for which we can prove locality (or construct equivalent local axiomatizations) using superposition – without having to explicitly take into account the congruence axioms.

Constrained clauses are often used in automated theorem proving in order to restrict the number of instances to be considered or for defining a notion of schematic saturation (cf. e.g. [17, 15, 16, 20]). In contrast with this type of results, the constraints we use here are formulae, equalities between variables (to which substitutions are applied) or more general so-called regular constraints. These constraints generalize regular expressions and allow infinite sets of clauses with a repeating structure to be captured by a single constrained clause. The use of the s^+ operation for reasoning about integer offsets in [20] is similar. The difference is that whereas in our work the regular expressions occur in constraints, in [20] they occur in the main clauses, and the constraints only ensure that some of the variables can only be instantiated with constants.

Structure of the paper. The paper is structured as follows: In Sect. 2 we introduce the terminology used in the paper and present the main results on local theory extensions. In Sect. 3 we give ways of recognizing locality and prove a locality transfer result for order-local theory extensions; then explain the problem we address in this paper and the idea of our solution. In Sect. 4 we define a constraint inference calculus which we use in Sect. 5 for giving finite saturations of infinite sets of clauses (where we also discuss the limitations of this approach). The full proofs are included in the extended version of this paper [8].

2 Preliminaries

In this section we introduce the terminology and main results used in the paper.

2.1 General Definitions

We build on the notions of [3, 21, 11] and shortly recall here the most important concepts concerning terms and orderings and the specific extensions (concerning constrained clauses) needed in this article. To keep the presentation concise, we restrict ourselves to single-sorted signatures. The many-sorted case works similarly, and it is explicitly taken into account in the later sections.

Terms and Clauses. Let $\Pi = (\Sigma, \mathsf{Pred})$ be a *signature* consisting of a set Σ of function symbols of fixed arity and a set Pred of predicate symbols of fixed arity, and let X be an infinite set of variables such that X and Σ are disjoint.

We denote by $\mathcal{T}_\Sigma(X)$ the set of all *terms* over Σ and X and by $\mathcal{T}_\Sigma$ the set of all *ground terms* over Σ. To improve readability, term tuples $(t_1, \ldots, t_n)$ will often be denoted by $\vec{t}$. The variables occurring in a term t or a term tuple $\vec{t}$ are denoted by $\mathrm{vars}(t)$ or $\mathrm{vars}(\vec{t})$, respectively.

An equation is a multiset of two terms $t_1, t_2 \in \mathcal{T}_\Sigma(X)$, usually written $t_1 \approx t_2$. A *predicative atom* is an expression of the form $P(t_1, \ldots, t_n)$, where $P \in \mathsf{Pred}$ is a predicate symbol of arity n and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma(X)$ are terms. An *atom* is an equation or a predicative atom. A *clause* is a pair of multisets of atoms, written $\Gamma \rightarrow \Delta$, interpreted as the conjunction of all atoms in the *antecedent* Γ implying the disjunction of all atoms in the *succedent* Δ. A clause is *Horn* if Δ contains at most one atom. If $C = A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ is a ground clause, we denote by $\neg C$ the set of unit Horn clauses $\rightarrow A_i$ and $B_j \rightarrow$.

Substitution Expressions. A *(basic) substitution* σ is a map from a finite set $X' \subseteq X$ of variables to $\mathcal{T}_\Sigma(X)$. The application of σ to a term t or a term tuple $\vec{t}$ is denoted by $t\sigma$ or $\vec{t}\sigma$, respectively. The substitution σ is *linear* if no variable occurs twice in the term set $\{x\sigma \mid x \in X'\}$. The *most general unifier* of two terms s, t is denoted by $\mathrm{mgu}(s, t)$.

Substitution expressions are built over substitutions and constructors $\circ$ (composition), $|$ (disjunction), and * (loop) of arity 2, 2 and 1, respectively. Substitution expressions are denoted as $\bar{\sigma}, \bar{\tau}$. The symbols $\circ$ and $|$ are written in infix notation, and * is written in postfix notation. We will often write $\bar{\sigma} \circ \bar{\tau}$ as $\bar{\sigma}\bar{\tau}$.

The *domain* $\mathrm{dom}(\bar{\sigma})$ and the *variable range* $\mathrm{VRan}(\bar{\sigma})$ of a substitution expression are defined as follows: For a substitution $\sigma : \{x_1, \ldots, x_n\} \rightarrow \mathcal{T}_\Sigma(X)$, we define $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$ and $\mathrm{VRan}(\sigma) = \mathrm{vars}(x_1\sigma, \ldots, x_n\sigma)$. For complex expressions, we have

$$\begin{array}{rlrl}
\mathrm{dom}(\bar{\sigma} \circ \bar{\tau}) &= \mathrm{dom}(\bar{\sigma}) & \mathrm{VRan}(\bar{\sigma} \circ \bar{\tau}) &= \mathrm{VRan}(\bar{\tau}) \\
\mathrm{dom}(\bar{\sigma}_1 | \bar{\sigma}_2) &= \mathrm{dom}(\bar{\sigma}_1) \cup \mathrm{dom}(\bar{\sigma}_2) & \mathrm{VRan}(\bar{\sigma}_1 | \bar{\sigma}_2) &= \mathrm{VRan}(\bar{\sigma}_1) \cap \mathrm{VRan}(\bar{\sigma}_2) \\
\mathrm{dom}(\bar{\sigma}^*) &= \mathrm{dom}(\bar{\sigma}) & \mathrm{VRan}(\bar{\sigma}^*) &= \mathrm{dom}(\bar{\sigma}) \cup \mathrm{VRan}(\bar{\sigma})
\end{array}$$

A substitution expression $\bar{\sigma}$ is *well-formed*, if (i) for each subexpression $\bar{\tau}_1 \circ \bar{\tau}_2$ of $\bar{\sigma}$, it holds that $\mathrm{VRan}(\bar{\tau}_1) = \mathrm{dom}(\bar{\tau}_2)$, (ii) for each subexpression $\bar{\tau}_1 | \bar{\tau}_2$ of $\bar{\sigma}$, it holds that $\mathrm{dom}(\bar{\tau}_1) = \mathrm{dom}(\bar{\tau}_2)$ and $\mathrm{VRan}(\bar{\tau}_1) = \mathrm{VRan}(\bar{\tau}_2)$, and (iii) for each subexpression $\bar{\tau}^*$ of $\bar{\sigma}$, it holds that $\mathrm{VRan}(\bar{\tau}) = \mathrm{dom}(\bar{\tau})$.

Constrained Clauses. A *constrained clause* $\alpha \,\|\, C$ consists of a clause C and a *regular constraint* α of the form $(x_1{\approx}y_1, \ldots, x_n{\approx}y_n)\bar{\sigma}$, also written as $(\vec{x}{\approx}\vec{y})\bar{\sigma}$, such that x_i, y_i are variables and $\bar{\sigma}$ is a well-formed substitution expression with domain $\{x_1, y_1, \ldots, x_n, y_n\}$. If a regular constraint α does not contain any equations, we call $\alpha \,\|\, C$ *unconstrained* and identify it with its clausal part C.

The application $(\alpha \,\|\, C)\sigma$ of a substitution to a constrained clause is defined as $\alpha\sigma \,\|\, C\sigma$. If $\alpha = (\vec{x}{\approx}\vec{y})\bar{\tau}$ is a regular constraint, then $\alpha\sigma$ is defined as $(\vec{x}{\approx}\vec{y})\bar{\tau}\sigma'$, where $\sigma' : \mathrm{VRan}(\bar{\tau}) \to \mathcal{T}(\Sigma, X)$ maps z to $z\sigma$ if $z \in \mathrm{dom}(\sigma)$ and to z otherwise.

The set of *ground instances* of a constrained clause $\alpha \,\|\, C$ consists of all ground clauses D for which there is a substitution σ such that $C\sigma = D$ and $\alpha\sigma$ is a satisfiable ground constraint. This means that regular constraints are interpreted syntactically.

Orderings. A (strict partial) *ordering* $\prec$ on a set S is a transitive and irreflexive binary relation on S. It is *total* if $s \prec t$ or $t \prec s$ whenever $s \neq t$. It is *well-founded* if there is no infinite descending chain $s_1 \succ s_2 \succ \ldots$ of elements of S.

An ordering $\prec$ on $\mathcal{T}_\Sigma(X)$ has the *subterm property* if $t \succ t'$ whenever t contains t' as a strict subterm. It is *stable under substitutions* if $t \prec t'$ implies $t\sigma \prec t'\sigma$ for all t, t' and all substitutions σ. It is a *reduction ordering* if it is well-founded, has the subterm property, and is stable under substitutions.

Let $\prec_\mathcal{T}$ be an ordering on $\mathcal{T}_\Sigma(X)$ and let $\prec$ be an ordering on atoms over $\mathcal{T}_\Sigma(X)$. Then $\prec$ is *compatible* with $\prec_\mathcal{T}$ if for all atoms A_1, A_2 it holds that $A_1 \prec A_2$ if every term in A_1 is bounded by a term in A_2, i.e. if for each term t_1 in A_1 there is a term t_2 in A_2 such that $t_1 \prec_\mathcal{T} t_2$. Note that for finite signatures of predicate symbols, any total ordering on atoms that is compatible with a total and well-founded term ordering is well-founded.

Any ordering $\prec$ on atoms can be extended to clauses in the following way. We consider clauses as multisets of occurrences of atoms. The occurrence of an atom A in the antecedent is identified with the multiset $\{A, A\}$; the occurrence of an atom A in the succedent is identified with the multiset $\{A\}$. Now we lift $\prec$ to atom occurrences as its multiset extension, and to clauses as the multiset extension of this ordering on atom occurrences.

An occurrence of an atom A in a clause C is *maximal* if there is no occurrence of an atom in C that is strictly greater with respect to $\prec$ than the occurrence of A. It is *strictly maximal* if there is no other occurrence of an atom in C that is greater than or equal to the occurrence of A w.r.t. $\prec$.

An (unconstrained) ground clause is *reductive (w.r.t. $\prec$)* if all of its $\prec$-maximal terms appear in the maximal atoms. A constrained clause is *reductive (w.r.t. $\prec$)* if all its ground instances are reductive (cf. [5]).

Denotations and Models. We define the *denotation* $[\![\bar{\sigma}]\!]$ of a substitution expression $\bar{\sigma}$ inductively as follows:

$$[\![\sigma]\!] = \{\sigma\} \qquad [\![\bar{\sigma}\bar{\tau}]\!] = \{\sigma\tau \mid \sigma \in [\![\bar{\sigma}]\!], \tau \in [\![\bar{\tau}]\!]\}$$
$$[\![\bar{\sigma}_1 | \bar{\sigma}_2]\!] = [\![\bar{\sigma}_1]\!] \cup [\![\bar{\sigma}_2]\!] \quad [\![\bar{\sigma}^*]\!] = \textstyle\bigcup_{n \geq 0} [\![\bar{\sigma}^n]\!]$$

Here $\bar{\sigma}^0$ denotes the substitution $\{x \mapsto x \mid x \in \mathrm{dom}\,\bar{\sigma}\}$, and $\bar{\sigma}^{n+1} = \bar{\sigma} \circ \bar{\sigma}^n$.

The semantics of the application of substitution expressions to terms and clauses and the semantics of constrained clause sets are defined just as one would expect by identifying a substitution expression with its denotation and by identifying a constrained clause $(\vec{x}{\approx}\vec{y})\bar{\sigma} \,\|\, C$ with the (potentially infinite) clause set $\{\vec{x}\sigma{\approx}\vec{y}\sigma \to C \mid \sigma \in [\![\bar{\sigma}]\!]\}$ (cf. [11] for details). An interpretation $\mathcal{I}$ is said to *model* a constrained clause set N, written $\mathcal{I} \models N$, if and only if $\mathcal{I} \models C$ for each C in the denotation of a constrained clause in N. In this case, $\mathcal{I}$ is called a *model* of N. A constrained clause set is *satisfiable* if it has a model.

Inferences and Redundancy. An *inference rule* is a relation on constrained clauses. Its elements are called *inferences* and written as

$$\frac{\alpha_1 \,\|\, C_1 \;\ldots\; \alpha_k \,\|\, C_k}{\alpha \,\|\, C} \quad .$$

The constrained clauses $\alpha_1 \,\|\, C_1, \ldots, \alpha_k \,\|\, C_k$ are called the *premises* and $\alpha \,\|\, C$ the *conclusion* of the inference. An *inference calculus* is a set of inference rules.

A constrained clause $(\vec{x}{\approx}\vec{y})\bar{\sigma} \,\|\, C$ is *redundant* w.r.t. a constrained clause set N if C is a tautology or if there is a variant $(\vec{x}{\approx}\vec{y})\bar{\tau} \,\|\, C$ of a constrained clause in N such that $[\![\bar{\sigma}]\!] \subseteq [\![\bar{\tau}]\!]$. An inference is called *redundant* w.r.t. N if its conclusion is redundant w.r.t. N or if a premise C is redundant w.r.t. $N \backslash \{C\}$. A constrained clause set N is *saturated* (w.r.t. a given inference calculus) if each inference with premises in N is redundant w.r.t. N.

A *derivation* is a finite or infinite sequence $N_0, N_1, \ldots$ such that for each i, there is an inference with premises in N_i and conclusion $(\vec{x}{\approx}\vec{y})\bar{\sigma} \,\|\, C$ that is not redundant w.r.t. N_i, such that $N_{i+1} = N_i \cup \{(\vec{x}{\approx}\vec{y})\bar{\sigma} \,\|\, C\}$.

2.2 Local Theory Extensions

Let $\Pi_0{=}(\Sigma_0, \mathsf{Pred})$ be a signature, and $\mathcal{T}_0$ be a theory with signature Π_0. We here consider extensions $\mathcal{T} := \mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$ with new function symbols Σ (called *extension functions*) whose properties are axiomatized using a set $\mathcal{K}$ of (universally closed) clauses in the extended signature $\Pi = (\Sigma_0 \cup \Sigma, \mathsf{Pred})$.

Example 1. *Let $\mathcal{T}_0$ be a theory of integers with signature containing the unary function s and the predicate symbol $\leq$. Let $\Sigma = \{f\}$ where f is a new function symbol. Let $\mathcal{K}_f^1 = \{\forall x, y\, (x{\leq}y \to f(x){\leq}f(y))\}$ and $\mathcal{K}_f^2 = \{\forall x (f(x){\leq}f(s(x)))\}$ (axiomatizations for the monotonicity of f). For $i = 1, 2$, $\mathcal{T}_i := \mathcal{T}_0 \cup \mathcal{K}_f^i$ is an extension of $\mathcal{T}_0$ with function f satisfying the set $\mathcal{K}_f^i$ of clauses.*

Our goal is to address proof tasks of the form $G \models_{\mathcal{T}_0 \cup \mathcal{K}} \perp$ (written also: $\mathcal{T}_0 \cup \mathcal{K} \cup G \models \perp$) where G is a set of ground clauses with additional (fresh) constants (in a countable set C), i.e. in the signature $\Pi^C = (\Pi_0 \cup \Sigma)^C = (\Sigma_0 \cup \Sigma \cup C, \mathsf{Pred})$.

Locality Conditions. Let $\mathcal{T}_0$ be an arbitrary theory with signature $\Pi_0 = (\Sigma_0, \mathsf{Pred})$, where the set of function symbols is Σ_0. Let $\Pi = (\Sigma_0 {\cup} \Sigma, \mathsf{Pred}) \supseteq \Pi_0$ be an extension by a non-empty set Σ of new function symbols and $\mathcal{K}$ be a set of (implicitly universally closed) clauses in the extended signature. Let C be a fixed countable set of fresh constants.

Notation: Let T be a set of ground terms in the signature Π^C. We denote by $\mathcal{K}[T]$ the set of all instances of $\mathcal{K}$ in which the terms starting with a function symbol in Σ are in T. Formally:

$\mathcal{K}[T] := \{\varphi\sigma \mid \forall\bar{x}.\, \varphi(\bar{x}) \in \mathcal{K},$ where (i) if $f \in \Sigma$ and $t = f(t_1, ..., t_n)$ occurs in $\varphi\sigma$ then $t \in T$; (ii) if x is a variable that does not appear below some Σ-function in φ then $\sigma(x) = x\}$.

An extension $\mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$ is *local* if it satisfies the following condition[1]:

(Loc) For every set G of ground clauses in Π^C it holds that
$\mathcal{T}_0 \cup \mathcal{K} \cup G \models \perp$ if and only if $\mathcal{T}_0 \cup \mathcal{K}[G] \cup G \models \perp$

where $\mathcal{K}[G] = \mathcal{K}[\mathsf{est}(\mathcal{K}, G)]$ consists of those instances of $\mathcal{K}$ in which the terms starting with *extension functions* are in the set $\mathsf{est}(\mathcal{K}, G)$ of extension ground terms (i.e. terms starting with a function in Σ) which already occur in G or $\mathcal{K}$. In [12] we generalized condition (Loc) by considering closure operators on ground terms. Let Ψ be a closure operator associating with every set T of ground terms a set $\Psi(T)$ of ground terms. For any set G of ground Π^C-clauses we write $\mathcal{K}[\Psi_{\mathcal{K}}(G)]$ for $\mathcal{K}[\Psi(\mathsf{est}(\mathcal{K}, G))]$. We define versions of locality in which the set of terms used in the instances of the axioms is described using the map Ψ:

(Loc^{Ψ}) For every set G of ground clauses in Π^C it holds that
$\mathcal{T}_0 \cup \mathcal{K} \cup G \models \perp$ if and only if $\mathcal{T}_0 \cup \mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G \models \perp$.

Extensions satisfying condition (Loc^{Ψ}) are called Ψ-local. A *finite* locality conditions (Loc_f^{Ψ}) is defined by restricting the locality conditions to hold for *finite* sets G of ground clauses.

Local theory extensions are Ψ-local, where Ψ=id (the identity operator). The order-local theories introduced in [5] satisfy a Ψ-locality condition, where T is a set of ground clauses $\Psi(T)$=$\{s \mid s$ ground term and $s \preceq t$ for some $t \in T\}$, where $\prec$ is an ordering on ground terms [5]. In this case we write also $\mathcal{K}[\preceq T]$ for $\mathcal{K}[\Psi(T)]$. If $T = \mathsf{st}(G)$ is the set of all ground terms occurring in G, we also use the notation $\mathcal{K}[\preceq G]$.

Hierarchical Reasoning. Let $\mathcal{T}_0 \subseteq \mathcal{T} = \mathcal{T}_0 \cup \mathcal{K}$ be a theory extension satisfying condition (Loc^{Ψ}). To check the satisfiability w.r.t. $\mathcal{T}$ of a set G of ground Π^C-clauses, we proceed as follows: By locality, $\mathcal{T} \cup G \models \perp$ iff $\mathcal{T}_0 \cup \mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G \models \perp$. We purify $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ by introducing, in a bottom-up manner, new constants c_t for subterms $t = f(g_1, \dots, g_n)$ with $f \in \Sigma$, g_i ground $(\Sigma_0 \cup C)$-terms, together with their definitions $c_t \approx t$. The set of formulae thus obtained has the form $\mathcal{K}_0 \cup G_0 \cup D$, where D consists of definitions of the form $c \approx f(g_1, \dots, g_n)$, where $f \in \Sigma$, c is a constant, $g_1, \dots, g_n$ are ground $(\Sigma_0 \cup C)$-terms, and $\mathcal{K}_0, G_0$ are Π_0^C-formulae. We reduce the problem to testing satisfiability in $\mathcal{T}_0$ as follows:

Theorem 2 ([18]). *Let $\mathcal{K}$ and G be as specified above. Assume that $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ satisfies condition (Loc^{Ψ}). Let $\mathcal{K}_0 \cup G_0 \cup D$ be obtained from $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ as explained above. Then $\mathcal{T}_0 \cup \mathcal{K} \cup G \models \perp$ if and only if $\mathcal{T}_0 \cup \mathcal{K}_0 \cup G_0 \cup \mathsf{Con}_0 \models \perp$*

(where $\mathsf{Con}_0 = \{ \bigwedge_{i=1}^{n} c_i \approx d_i \rightarrow c = d \mid \begin{array}{l} c \approx f(c_1, \dots, c_n) \in D, \\ d \approx f(d_1, \dots, d_n) \in D \end{array} \}$*).*

If $\mathcal{K}[\Psi_{\mathcal{K}}(G)]$ is finite and $\mathcal{K}_0 \cup G_0 \cup \mathsf{Con}_0$ belongs to a decidable fragment of $\mathcal{T}_0$ then we can effectively check the decidability of G.

[1] It is easy to check that the formulation we give here and that in [18] are equivalent.

3 Recognizing Ψ-local Theory Extensions

We present several ways of recognizing local and Ψ-local theory extensions.

3.1 Locality and Embedability

Links between *locality of a theory* and *embeddability* of partial models into total ones were established in [6]. Similar results can also be obtained for *local theory extensions*. When establishing links between locality and embeddability, we require that the extension clauses in $\mathcal{K}$ are *flat* and *linear* w.r.t. Σ-functions:

A *(non-ground) extension clause* D is *Σ-flat* when all symbols below a Σ-function symbol in D are variables. D is *Σ-linear* if, whenever a variable occurs in two terms of D which start with Σ-functions, the terms are identical, and no such term contains two occurrences of a variable.

Let $\Pi = (\Sigma, \mathsf{Pred})$ be a first-order signature with set of function symbols Σ and set of predicate symbols Pred. A *partial Π-structure* is a structure $\mathcal{A} = (A, \{f_{\mathcal{A}}\}_{f\in\Sigma}, \{P_{\mathcal{A}}\}_{P\in\mathsf{Pred}})$, where A is a non-empty set, for every $f \in \Sigma$ with arity n, $f_{\mathcal{A}}$ is a partial function from A^n to A, and for every $P \in \mathsf{Pred}$, $P_{\mathcal{A}} \subseteq A^n$. We consider constants (0-ary functions) to be always defined. $\mathcal{A}$ is called a *total structure* if the functions $f_{\mathcal{A}}$ are all total. Given a (total or partial) Π-structure $\mathcal{A}$ and $\Pi_0 \subseteq \Pi$ we denote the reduct of $\mathcal{A}$ to Π_0 by $\mathcal{A}|_{\Pi_0}$. (For the precise definition of a weak partial model for a set of clauses see e.g. [18, 13].) If $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{K}$ is an extension of a Π_0-theory $\mathcal{T}_0$ with new function symbols in Σ and clauses $\mathcal{K}$, we denote by $\mathsf{PMod}^{\Psi}_w(\Sigma, \mathcal{T})$ the set of weak partial models $\mathcal{A}$ of $\mathcal{T}$ whose Σ_0-functions are total, and all terms in $\Psi(\mathsf{est}(\mathcal{K}) \cup \{f(a_1, \dots, a_n) \mid f \in \Sigma, f_{\mathcal{A}}(a_1, \dots, a_n)$ is defined$\}$ are defined – in the extended structure $\mathcal{A}^A$ with constants from $\mathcal{A}$. In [18, 12, 13] we considered embeddability properties of partial algebras, e.g. (Emb^{Ψ}_w) and $(\mathsf{Emb}^{\Psi}_{w,f})$.

(Emb^{Ψ}_w) Every $\mathcal{A} \in \mathsf{PMod}^{\Psi}_w(\Sigma, \mathcal{T})$ weakly embeds into a total model of $\mathcal{T}$.

Condition $(\mathsf{Emb}^{\Psi}_{w,f})$ requires embeddability only for partial algebras where the extension functions have a *finite* domain of definition. We proved that if $\mathcal{K}$ is a set of Σ-flat and Σ-linear clauses in the signature Π and all weak partial models of an extension $\mathcal{T}_0 \cup \mathcal{K}$ of a base theory $\mathcal{T}_0$ with total Σ_0-functions can be embedded into a total model of the extension, then the extension is local (i.e. that (Emb^{Ψ}_w) implies (Loc^{Ψ})) [18, 12, 13]. Conversely, we showed that if $\mathcal{K}$ is a set of Σ-flat clauses in the signature Π then if $\mathcal{T}_0$ is a first-order theory and the extension $\mathcal{T}_0 \subseteq \mathcal{T}{=}\mathcal{T}_0{\cup}\mathcal{K}$ satisfies (Loc^{Ψ}) then every model in $\mathsf{PMod}^{\Psi}_w(\Sigma, \mathcal{T})$ weakly embeds into a total model of $\mathcal{T}$.

Example 3. *Let $\mathcal{T}_0$ be the theory of integers with successor (s) and ordering $\leq$ described by the model $(\mathbb{N}, s, \leq)$. Let f be a new function symbol.*

- $\mathcal{K}^1_f = \{x \leq y \rightarrow f(x) \leq f(y)\}$ *satisfies condition* $\mathsf{Emb}^{\mathsf{id}}{}_w$ *hence defines a local theory extension.*

- *Neither* $\mathcal{K}_f^2 = \{f(x) \leq f(s(x))\}$ *nor its flattened version* $\{y \approx s(x) \rightarrow f(x) \leq f(y)\}$ *satisfy* $\mathsf{Emb}_{\mathsf{w}}^{\mathsf{id}}$ *(there are weak partial models of this axiom which cannot be extended to total ones – for instance the partial model* $\mathcal{A}$ *with support* $\mathbb{N}$ *for which* $f_{\mathcal{A}}(2) = 4, f_{\mathcal{A}}(4) = 2$ *and* f *is undefined everywhere else).*

3.2 Locality and Saturation

In [4, 5], Basin and Ganzinger defined the notion of *order locality*, established a link between peak saturation and order locality, and used these results for automated complexity analysis. Given a term ordering $\prec$, we say that a set $\mathcal{K}$ of clauses entails a clause C bounded by $\prec$ (notation: $\mathcal{K} \models_{\preceq} C$), if and only if there is a proof of $\mathcal{K} \models C$ from those ground instances of clauses in $\mathcal{K}$ in which (under $\prec$) each term is smaller than or equal to some term in C.

We can also consider an ordered hyperresolution calculus for Horn clauses with a selection function which selects in every clause C the set of all negative atoms of C which contain a term which is maximal in C w.r.t. $\prec$ [5]. In [4, 5] the following terminology is used in this case: *Negative* (resp. *positive*) premises are premises in which the literal resolved upon is negative (resp. positive). *Peak inferences* are inferences with the property that for every term t in the conclusion there is a larger term $t' \succ t$ in the negative premise. *Plateau inferences* are inferences for which in the succedent of a negative premise there exists an occurrence of a maximal term. A set of Horn clauses $\mathcal{K}$ is *peak saturated* if all peak inferences are redundant for which (i) the second, negative premise is in $\mathcal{K}$, and (ii) the first, positive premise's antecedent does not contain a maximal term, and (iii) the positive premise is in $\mathcal{K}$ or generated from $\mathcal{K}$ using plateau inferences.

A set of clauses $\mathcal{K}$ is called *order local* w.r.t. $\prec$ if whenever $\mathcal{K} \models C$ for a ground clause C, then $\mathcal{K} \models_{\preceq} C$.

Theorem 4 ([4, 5]). *Let* $\prec_{\mathcal{T}}$ *be a well-founded (possibly partial) term ordering and* $\prec$ *a compatible and total atom ordering in first-order logic without equality. Let* $\mathcal{K}$ *be a set of clauses which is reductive w.r.t.* $\prec_{\mathcal{T}}$*.*

- *If* $\mathcal{K}$ *is saturated w.r.t.* $\prec$*-ordered resolution, then* $\mathcal{K}$ *is order local w.r.t.* $\prec$*.*
- *Let* $\mathcal{K}$ *be a set of Horn clauses.* $\mathcal{K}$ *is peak saturated w.r.t.* $\prec$*-ordered hyperresolution with selection (cf. [5]) if and only if* $\mathcal{K}$ *is order local w.r.t.* $\prec$*.*

We now consider a set of clauses of the form $N = N_0 \cup \mathcal{K}$, where N_0 is a set of clauses over a signature Π_0 – an axiomatization of a base theory $\mathcal{T}_0$ – and $\mathcal{K}$ is a set of $\Pi_0 \cup \Sigma$-clauses, all containing additional extension functions in a set Σ.

Theorem 5. *Assume that* $N = N_0 \cup \mathcal{K}$ *is a set of clauses where* N_0 *consists of* Π_0*-clauses and* $\mathcal{K}$ *consists of* $\Pi_0 \cup \Sigma$*-clauses which contain extension functions in* Σ*. Let* $\prec$ *be an ordering with the property that terms starting with a* Σ*-symbol are larger than all other terms. Assume that* $N = N_0 \cup \mathcal{K}$ *is order local. Then:*

(1) N_0 *is order local.*
(2) The extension $N_0 \subseteq N_0 \cup \mathcal{K}$ *is* Ψ*-local for a suitable* Ψ*.*

(3) Assume that $\mathcal{T}_1$ is a definitional extension of the theory axiomatized by N_0 (i.e. obtained by extending the signature with new predicate symbols in a set Pred' *with properties axiomatized by formulae of the form $R(x_1, \ldots, x_n) \leftrightarrow \phi_R(x_1, \ldots, x_n)$, where ϕ_R is a Π_0-formula). Then the extension $\mathcal{T}_1 \subseteq \mathcal{T}_1 \cup \mathcal{K}$ satisfies a Ψ'-locality condition for all sets of clauses in the signature $\Pi_0 \cup \Sigma$.*

Proof. (1) follows from the locality of $N_0 \cup \mathcal{K}$ and the fact that if C is a Π_0-clause then $N_0 \cup \mathcal{K} \models_{\preceq} C$ iff $N_0 \models_{\preceq} C$. (2) By locality, the following are equivalent: (i) $N_0 \cup K \models C$, (ii) $(N_0 \cup K)[\preceq C] \models C$, (iii) $(N_0 \cup K)[\preceq_\Sigma C] \models C$ (where $\mathcal{K}[\preceq_\Sigma C]$ is the set of instances of $\mathcal{K}$ in which the terms starting with a symbol in Σ are instantiated with ground terms smaller than some ground term of C). (3) is proved using the link between embeddability and locality; the result holds for all theories $\mathcal{T}_1$ with the property that on every model P of N_0 interpretations of the relations in Pred' can be defined such that P becomes a model of $\mathcal{T}_1$. □

Theorem 4 allows us to obtain, by saturation, local axiomatizations from non-local ones. We can saturate $\mathcal{K}$ under peak redundancy by first adding all clauses obtained by plateau inferences between clauses in this set (obtaining a set P), and then the conclusions of all peak inferences with negative premise in $\mathcal{K}$ and positive premises of the corresponding form which are in $\mathcal{K}$ or P. One drawback is that equality cannot be used as a built-in predicate: If the clauses contain the equality predicate then the congruence axioms have to be added explicitly, which can be inefficient. Another drawback is that the size of the saturated sets of clauses can be very large. Often, in fact, infinitely many clauses are generated.

Example 6. *Consider* $\mathsf{Pre} \cup \{f(x) \leq f(s(x))\}$*, where* $\mathsf{Pre} = \{x \leq x,\ \ x \leq y \wedge y \leq z \rightarrow x \leq z\}$*. By saturation we obtain the infinite set*[2]*:* $\{f(x) \leq f(s^n(x)) \mid n \geq 0\} \cup \mathsf{Pre}$*. In such cases, a usual resolution-based theorem prover will not be able to detect saturation, because the set of clauses which are generated is infinite.*

Our goal is to obtain finite representations of possibly infinite sets of clauses. For this, we will use constrained clauses. As a by-product, the form of the constraints will allow us to (conservatively) extend the language – e.g. by defining new predicates – in order to obtain local presentations

4 A Constrained Inference Calculus

We use the standard inference rules for constrained ordered resolution and superposition (cf. [11]). The ordered resolution rule, for example, is:

Ordered Resolution:

$$\frac{\alpha_1 \,\|\, \Gamma_1 \rightarrow \Delta_1, A_1 \quad \alpha_2 \,\|\, \Gamma_2, A_2 \rightarrow \Delta_2}{(\alpha_1, \alpha_2 \,\|\, \Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2)\sigma}$$

where (1) σ is the most general unifier of A_1 and A_2, (2) $A_1\sigma$ is strictly maximal in $(\Gamma_1 \rightarrow \Delta_1, A_1)\sigma$ and (3) $A_2\sigma$ is maximal in $(\Gamma_2, A_2 \rightarrow \Delta_2)\sigma$.

[2] Similarly if we consider the flattened version of $\mathcal{K}_f^2$: $\{y = s(x) \rightarrow f(x) \leq f(y)\}$.

This inference system can as usual be restricted by means of a literal selection function. It is sound and complete as a slight variation of [10, 9], provided that constraint satisfiability is decidable. As an example, this is the case when the constraints are increasing (cf. Theorem 8 below).

We now extend the calculus to an inference system including a *melting* rule [11], which serves as a limited form of induction. To define melting, we need the notion of an ancestor. In any of the usual inferences, the *ancestors* of the conclusion are the rightmost premise and all of its ancestors.

Melting:

$$\frac{(\vec{x}{\approx}\vec{y})\bar{\sigma} \parallel C \quad (\vec{x}{\approx}\vec{y})\bar{\sigma}\bar{\tau}' \parallel C'}{(\vec{x}{\approx}\vec{y})\bar{\sigma}'' \parallel C}$$

where (1) $(\vec{x}{\approx}\vec{y})\bar{\sigma} \parallel C$ is an ancestor of $(\vec{x}{\approx}\vec{y})\bar{\sigma}\bar{\tau}' \parallel C'$, and (2) $(\vec{x}{\approx}\vec{y})\bar{\sigma}\bar{\tau}' \parallel C'$ is a variant of $(\vec{x}{\approx}\vec{y})\bar{\sigma}\bar{\tau} \parallel C$, and either (3.i) $\bar{\sigma}$ is of the form $\bar{\sigma} = \bar{\sigma}_1\bar{\sigma}_2^*$ and $\bar{\sigma}'' = \bar{\sigma}_1(\bar{\sigma}_2|\bar{\tau})^*$, or (3.ii) $\bar{\sigma}$ is not of this form and $\bar{\sigma}'' = \bar{\sigma}\bar{\tau}^*$.

The ancestors of the conclusion of a melting inference are defined as the ancestors of the leftmost premise.

Intuitively, the melting rule states: if it is possible to derive $\alpha\bar{\tau} \parallel C$ from $\alpha \parallel C$, then it is also possible to repeat this process to derive $\alpha\bar{\tau}\bar{\tau} \parallel C$ and so on. This is not always true, but it does hold whenever there are no inferences where both premises are constrained (except for melting inferences). Cf. [11, 7] for more details.

Melting is important because it allows us in many cases to saturate clause sets much faster, or even to turn an infinite saturation into a finite one.

Example 7. *Consider the clause set $\{x{\approx}y \parallel P(x) \to Q(y),\ P(x) \to P(s(x))\}$, where the ordering is chosen such that $P(t_1) \succ Q(t_2)$ for all ground terms t_1, t_2. When saturating this clause set, we successively derive all clauses of the form $s^n(x){\approx}y \parallel P(x) \to Q(y)$ by iterating the following inference step:*

$$\frac{P(x) \to P(s(x)) \quad s^n(x){\approx}y \parallel P(x) \to Q(y)}{s^{n+1}(x){\approx}y \parallel P(x) \to Q(y)}$$

This derivation does not terminate. With melting however, we can make make one such inference to derive $s(x){\approx}y \parallel P(x) \to Q(y)$ and directly follow up with a melting inference:

$$\frac{x{\approx}y \parallel P(x) \to Q(y) \quad s(x){\approx}y \parallel P(x) \to Q(y)}{s^*(x){\approx}y \parallel P(x) \to Q(y)}$$

(where $s^*(x){\approx}y$ stands for $(x{\approx}y)\{x \mapsto s(x), y \mapsto y\}^*$). After this inference, the clause set is already saturated.

However, note that **Melting** does not make the inference system terminating in general. In [11], we presented conditions under which termination results for saturation on unconstrained nonequational clauses carry over to constrained clauses. Extending these results to clauses containing equational literals will be the subject of further work.

5 Locality and Melting Constraints

In many settings, the regular constraints that appear just stack increments on both sides of an equation. This is for example the case with the strict monotonicity[3] axiom $s(x)\approx y \to s(f(x)) \leq f(y)$, which gives rise to clauses of the form $s^n(x)\approx y \to s^n(f(x)) \leq f(y)$ for each $n \geq 0$, or to the constrained clause

$$(x\approx y, v\approx w)\sigma^* \parallel v\approx f(x) \to w \leq f(y)$$

for $\sigma = \{x \mapsto s(x), y \mapsto y, v \mapsto s(v), w \mapsto w\}$. If $f : S_1 \to S_2$ is a function symbol that connects two different sorts (which have different successor functions s_1, s_2), then σ would be $\sigma = \{x \mapsto s_1(x), y \mapsto y, v \mapsto s_2(v), w \mapsto w\}$.

Let $\bar{\sigma}$ be a substitution expression over Σ. Let Σ contain for each sort S of a domain element of $\bar{\sigma}$ a unique unary function symbol $s_S : S \to S$. Then $\bar{\sigma}$ is *increasing for Σ* if, for each basic substitution τ in σ and each variable x, either $\tau(x)$ is a constant or $\tau(x) = s_S^k(x)$ for some $k \geq 0$, where S is the sort of x. A regular constraint is *increasing for Σ* if it is empty or if its substitution expression is of the form $\bar{\sigma}_1 \circ \cdots \circ \bar{\sigma}_n$ such that each $\bar{\sigma}_i$ is increasing for some $\Sigma_i \subseteq \Sigma$, and $\Sigma_i \cap \Sigma_j$ contains only constants for $i \neq j$.

We use a shorthand notation for increasing substitution expressions and increasing constraints: For example, if $\sigma = \{x \mapsto s(s(x)), y \mapsto y\}$ and $\tau = \{x \mapsto 0, y \mapsto 0\}$, we write $\sigma^*\tau$ as $(s^2, s^0)^*(0,0)$, and we write the constraint $(x\approx y)\sigma^*\tau$ as $(x\approx y)(s^2, s^0)^*(0,0)$. With this notation, the constrained clause describing strict monotonicity becomes $(x\approx y, v\approx w)(s^1, s^0, s^1, s^0)^* \parallel v\approx f(x) \to w \leq f(y)$, or

$$(x\approx y, v\approx w)(s_1^1, s_1^0, s_2^1, s_2^0)^* \parallel v\approx f(x) \to w \leq f(y)$$

if the domain and range of f have different sorts and we have two successor functions. If in addition the substitution operates independently on the different variables of the constraint, we simplify the notation even more and write, for example, $x\approx s^*(s(y))$ for $(x\approx y)\{x \mapsto x, y \mapsto s(y)\}\{x \mapsto x, y \mapsto s(y)\}^*$.

Increasing constraints are one class where the constrained calculus is applicable:

Theorem 8. *Let Σ be a set of function symbols and α an increasing regular constraint for Σ. The satisfiability of α is decidable.*

Proof. The proof proceeds by rewriting the substitution expression of α until the only remaining loops are of the form $(s_1^k, s_2^0, \ldots, s_n^0)^*$ and loops over the same unary function symbols cannot interact. At this point, satisfiability reduces to an easy unification problem. □

Example 9. *Examples of clause sets where the use of constraints comes in handy:*

- *Let all function symbols s_i be unary. For subterm locality, $N[\preceq G]$ is equivalent to the set of clauses of the form*

$$(s_1|\ldots|s_n)^*(x_1)\approx t_1, \ldots, (s_1|\ldots|s_n)^*(x_m)\approx t_m, \alpha \parallel C\ ,$$

[3] Similar considerations also hold for the monotonicity axiom $s(x)\approx y \to f(x) \leq f(y)$.

where $\alpha \parallel C \in N$ has the free variables $x_1, \ldots, x_m$ and $t_1, \ldots, t_m$ are maximal terms in G (w.r.t. the subterm ordering). That will be much more concise than writing down all of $N[\preceq G]$ because the number of instances goes down from $|\mathsf{st}(G)|^m$ to $|\mathsf{maxst}(G)|^m$ ($\mathsf{maxst}(G)$ are the maximal subterms of G).

- *The standard ordering over the naturals is completely described by the saturated set $\{y{\approx}s^+(x) \parallel x < y,\ y{\approx}s^+(x) \parallel y \not< x\}$.*
- *Cycle-free lists can be characterized without a reachability predicate just by the clause $y{\approx}p^*(p(x)) \parallel x{\approx}y \to x{\approx}nil$. More generally, a similar clause characterizes absolutely free unary constructors.*

As mentioned before, Theorem 4, a result by Basin and Ganzinger [5], does not hold for full superposition instead of ordered resolution. Theorem 10 below shows how a slightly restricted form of superposition allows to recover locality, and that this is even true for clauses with regular constraints. Our proof extends the one from [5].

In the following, saturation can always be interpreted as saturation with or without melting, because (due to the definition of ground clauses) the constraints do not affect the proofs.

Theorem 10. *Let $\prec_{\mathcal{T}}$ be a reduction ordering and $\prec$ a compatible and total atom ordering. Let N be a set of constrained clauses that is reductive w.r.t. $\prec_{\mathcal{T}}$ and saturated w.r.t. $\prec$. Let each constrained clause in N with positive equational atoms contain either a unique positive equation which is also maximal, or a negative equation which is maximal. Let C be a ground clause whose succedent does not contain any equations. Then $N \models C$ iff $N \models_{\preceq} C$.*

Proof. The direction $N \models_{\preceq} C \implies N \models C$ is obvious and independent of the properties of N and C. For the opposite direction, we prove by induction on the length of this proof that (i) all clauses that are used in the proof are $\preceq C$, (ii) all clauses that are derived in the proof contain equational atoms only negatively, and (iii) all clauses that are derived in the proof are unconstrained ground clauses. This implies the statement of the theorem. □

In some cases, saturation is still too strong a requirement to arrive at finite local axiomatizations.

Example 11. *Consider the theory of a strictly monotone function f as described in the beginning of this section, together with the theory of preorders:*

$$N_{\leq,f} = \{x \leq x, \quad x \leq y \wedge y \leq z \to x \leq z, \quad \alpha \parallel v{\approx}f(x) \to w \leq f(y)\}$$

For now, we ignore the exact shape of the constraint α. The axiomatization $N_{\leq,f}$ is local, but we cannot show this using the previous theorem, because inferences between the clauses for transitivity and monotonicity allow the derivation of additional clauses of the form

$$\alpha \parallel v{\approx}f(x), w_1 \leq w_2 \leq \ldots \leq w \to w_1 \leq f(y)\ .$$

When the theory does not contain equality literals, this can be remedied by considering a straightforward extension of peak saturation to constrained clauses:

An inference $\alpha_1 \,\|\, C_1,\ \alpha_2 \,\|\, C_2 \vdash \alpha \,\|\, C$ between constrained clauses is called a *peak inference* (resp. *plateau inference*) if the unconstrained inference $C_1, C_2 \vdash C$ is a peak inference (resp. plateau inference). A set of constrained Horn clauses N is *peak saturated* if all peak inferences are redundant for which (i) the second, negative premise is in N and (ii) the first, positive premise's antecedent does not contain a maximal term, and (iii) the positive premise is in N or generated from N using plateau inferences.

Theorem 12. *Let $\prec_{\mathcal{T}}$ be a well-founded term ordering and $\prec$ a compatible and total atom ordering. Let N be a set of constrained Horn clauses without equality that is reductive w.r.t. $\prec_{\mathcal{T}}$ and peak saturated w.r.t. $\prec$. Let C be a ground clause without equality. Then $N \models C$ iff $N \models_{\preceq} C$.*

Proof. The proof of the respective theorem for unconstrained clauses in [5] only makes use of properties of ground inferences. Because the ground instances of constrained clauses are defined in such a way that they are unconstrained, that proof carries over to constrained clause sets word by word. Note that for constrained clauses without equality literals, the only applicable rules are ordered resolution and factoring. □

Example 13. *In our running example of strict monotonicity, the only nonredundant inference that is enabled for $N_{\leq,f}$ is between the constrained clause for monotonicity and an instance of transitivity:*

$$\frac{\alpha \,\|\, v{\approx}f(x) \to w \leq f(y) \qquad \|\, w' \leq w \wedge w \leq f(y) \to w' \leq f(y)}{\alpha \,\|\, v{\approx}f(x), w' \leq w \to w' \leq f(y)}$$

(There is also an inference into the other antecedent literal of the transitivity clause, for which the situation is similar.) It is a plateau inference because $f(y)$ appears in the succedent of the right premise. The inferences from this clause and transitivity are again plateau inferences, and the same holds for all further inferences. Since no peak inferences are possible, $N_{\leq,f}$ is peak-saturated. By Theorem 12, $N_{\leq,f}$ is also order local.

Example 14. *Other examples where the theorem directly proves locality because the constrained clause sets are (peak) saturated:*

- *The theory axiomatized by: $\{\ x{\approx}s^*(0) \,\|\, P(x)\}$*
- *The theory of even and odd numbers axiomatized as follows:*

$$x{\approx}(ss)^*(0) \,\|\, \mathit{Even}(x), \quad x{\approx}(ss)^*(s(0)) \,\|\, \mathit{Odd}(x)\ \}$$

- *The theory of number pairs with an even sum (cf. [5]):*

$$\begin{aligned}\{x{\approx}(ss)^*(0) \quad &, y{\approx}(ss)^*(0) \quad \,\|\, \mathit{Evensum}(x,y),\\ x{\approx}(ss)^*(s(0)), &\, y{\approx}(ss)^*(s(0)) \,\|\, \mathit{Evensum}(x,y)\ \}\end{aligned}$$

- *Monotonicity on intervals* $\{a_i, \ldots, b_i\}$*,* $i \in I$*, where the* a_i *and* b_i *are concrete natural numbers:*

$$N_{\leq} \cup \bigcup_{i \in I} \{\ s^*(a_i) \approx x, s^*(x) \approx y, s^*(y) \approx b_i \parallel f(x) \leq f(y)\ \}$$

- *Monotonicity on unbounded intervals like* $\{a, \ldots\}$ *and* $\{\ldots, b\}$ *(similar).*

Example 15. *Consider the theory axiomatized by* $N = N_0 \cup \{\ x \approx s^*(0) \parallel P(x)\}$*, where* N_0 *is the axiomatization of natural numbers consisting of the absolutely free constructor axioms for* s *and* 0 *(a set of Horn clauses which is local [19] and can be proved to be order local also using Theorem 12). Let* $C = P(a)$*, where* a *is a new constant such that* $0 \prec a \prec s(0)$*. By locality, we know that* $N \models C$ *iff* $N[\preceq C] \models C$*. Then* $N[\preceq C] = \{P(0)\}$*, so* $N[\preceq C] \cup \neg C$ *is satisfiable. This shows that* $N \not\models C$*. (This is explained by the fact that we do not consider satisfiability in the initial model of the natural numbers, but in* some *model of* N*.)*

Often the axiomatizations have the form $N = N_0 \cup \mathcal{K}$, where N_0 is an axiomatization of a base theory and $\mathcal{K}$ defines properties of additional functions. The constraints in the saturated form may suggest ways of defining new predicates which would allow to use a finite clause notation. Assume f has sort $i \to o$. The clause $s^*(x) \approx y \parallel f(x) \leq f(y)$ can be replaced for instance with $x \leq' y \to f(x) \leq f(y)$, where $\leq'$ is a new predicate. The form of the constraint guides in giving a (possibly recursive) definition for $\leq'$. If the extension $\mathcal{T}_1$ obtained this way has one of the properties in Theorem 5(3), then the extension $T_1 \subseteq T_1 \cup \mathcal{K}$ satisfies a suitable Ψ'-locality property.

Limitations. For computing finite saturations (or peak saturations) of sets of constrained clauses it is very important to have a decision procedure for checking satisfiability of the constraints. In Theorem 8 we showed that if the constraints only contain unary constructors, then checking satisfiability is decidable. However, satisfiability is undecidable for most extensions of the presented fragment of regular constraints. This can usually be proven by a reduction of the Post correspondence problem.

Theorem 16. *Satisfiability of regular constraints with substitution expression* σ *is undecidable, even if all function symbols are at most unary and* σ *satisfies all conditions of an increasing substitution expression except the first.*

Proof. We show that with every Post correspondence problem P we can associate a regular constraint α with substitution expression σ (which satisfies all conditions of an increasing substitution expression except the first) such that P has a solution if, and only if, α is satisfiable. □

6 Conclusion

In this paper we presented a method for obtaining finite representations of local sets of clauses from possibly non-local ones. We extended the work of Basin and Ganzinger [4, 5]: In order to address the fact that saturation can generate infinite sets of clauses, we used constrained clauses, which allow us to give a finite representation for possibly infinite saturated sets of clauses and defined an ordered resolution and superposition calculus for such constrained clauses. We established links between locality and saturation for constrained clauses. The form of the constraints suggests definitions for new predicates which would yield saturated – hence local – theory axiomatizations.

In future work we would like to analyze possibilities of reasoning in the initial model (not investigated here), and ways of defining new predicate symbols, e.g. using recursive definitions; we hope that some of the results in [19] could prove useful for this. We plan to study the applicability of our results to reasoning in various data structures.

Acknowledgments. 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] Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Trans. Comput. Log. 10(1) (2009)

[2] Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Inf. Comput. 183(2), 140–164 (2003)

[3] Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. J. of Logic and Computation 4(3), 217–247 (1994)

[4] Basin, D., Ganzinger, H.: Complexity analysis based on ordered resolution. In: Proc. 11th IEEE Symposium on Logic in Computer Science (LICS 1996), pp. 456–465. IEEE Computer Society Press (1996)

[5] Basin, D.A., Ganzinger, H.: Automated complexity analysis based on ordered resolution. Journal of the ACM 48(1), 70–109 (2001)

[6] Ganzinger, H.: Relating semantic and proof-theoretic concepts for polynomial time decidability of uniform word problems. In: Proc. 16th IEEE Symposium on Logic in Computer Science (LICS 2001), pp. 81–92. IEEE Computer Society Press (2001)

[7] Horbach, M.: Superposition-based Decision Procedures for Fixed Domain and Minimal Model Semantics. PhD thesis, Max Planck Institute for Computer Science and Saarland University (2010)

[8] Horbach, M., Sofronie-Stokkermans, V.: Obtaining finite local theory axiomatizations via saturation. Technical Report ATR 93, Sonderforschungsbereich/Transregio 14 AVACS (2013)

[9] Horbach, M., Weidenbach, C.: Superposition for fixed domains. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 293–307. Springer, Heidelberg (2008)

[10] Horbach, M., Weidenbach, C.: Decidability results for saturation-based model building. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 404–420. Springer, Heidelberg (2009)

[11] 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)

[12] 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)

[13] Ihlemann, C., Sofronie-Stokkermans, V.: On hierarchical reasoning in combinations of theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 30–45. Springer, Heidelberg (2010)

[14] Kirchner, H., Ranise, S., Ringeissen, C., Tran, D.-K.: On superposition-based satisfiability procedures and their combination. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 594–608. Springer, Heidelberg (2005)

[15] Lynch, C., Morawska, B.: Automatic decidability. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), pp. 7–16. IEEE Comp. Soc. (2002)

[16] Lynch, C., Ranise, S., Ringeissen, C., Tran, D.-K.: Automatic decidability and combinability. Inf. Comput. 209(7), 1026–1047 (2011)

[17] Nieuwenhuis, R., Rubio, A.: Theorem proving with ordering constrained clauses. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 477–491. Springer, Heidelberg (1992)

[18] 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)

[19] Sofronie-Stokkermans, V.: Locality results for certain extensions of theories with bridging functions. In: Schmidt, R.A. (ed.) CADE-22. LNCS (LNAI), vol. 5663, pp. 67–83. Springer, Heidelberg (2009)

[20] Tushkanova, E., Ringeissen, C., Giorgetti, A., Kouchnarenko, O.: Automatic decidability: A schematic calculus for theories with counting operators. In: Proceedings the RTA 2013 (to appear, 2013)

[21] Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 2, ch. 27, pp. 1965–2012. Elsevier (2001)

Non-cyclic Sorts for First-Order Satisfiability

Konstantin Korovin*

School of Computer Science
The University of Manchester
United Kingdom
korovin@cs.man.ac.uk

Abstract. In this paper we investigate the finite satisfiability problem for first-order logic. We show that the finite satisfiability problem can be represented as a sequence of satisfiability problems in a fragment of many-sorted logic, which we call the non-cyclic fragment. The non-cyclic fragment can be seen as a generalisation of the effectively propositional fragment (EPR) in the many-sorted setting. We show that the non-cyclic fragment is decidable by instantiation-based methods and present a linear time algorithm for checking whether a given clause set is in this fragment. One of the distinctive features of our finite satisfiability translation is that it avoids unnecessary flattening of terms, which can be crucial for efficiency. We implemented our finite model finding translation in iProver and evaluated it over the TPTP library. Using our translation it was possible solve a large class of problems which could not be solved by other systems.

1 Introduction

Currently, the most successful methods for finite model finding in first-order logic are based on exhaustive flattening of function terms [3, 10]. We argue that flattening can have a detrimental effect on efficiency and present a new translation of the finite model finding problem into a fragment of many-sorted logic, which we call *the non-cyclic fragment*.

One of the main properties of the non-cyclic fragment is that it has a finite Herbrand universe and therefore can be seen as a natural generalisation of the EPR fragment (or Bernays-Shönfinkel-Ramsey fragment) in the many-sorted setting. The non-cyclic fragment is defined by imposing a condition on the signature, which prohibits cyclic dependencies between functions. When this condition is satisfied we call the signature non-cyclic. Variants of this fragment have been investigated in a different context in [1, 12, 14]. In this paper we take an algorithmic point of view. First, we argue that instantiation-based reasoning methods such as Inst-Gen [15, 20] and Model Evolution [6] are decision procedures for the non-cyclic fragment. Second, we present a linear time algorithm for checking whether a many-sorted signature is non-cyclic and more generally for classifying sorts into cyclic and non-cyclic sorts. Let us note that in a number of applications such as hardware verification and knowledge representation, signatures can contain many thousands of symbols and hence we need efficient algorithms for checking whether a signature is non-cyclic.

* Supported by a Royal Society University Fellowship.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 214–228, 2013.

We observe that in many cases problems do not fall completely into the EPR or more generally non-cyclic fragment, but rather contain combinations of EPR/non-cyclic sorts with cyclic sorts. We propose to take advantage of this sort separation in the setting of finite model finding. For this we extended a framework of the EPR-based finite model finding developed in [3]. The translation in [3] is based on exhaustive flattening of terms which allows one to replace functions with predicates. Exhaustive flattening of terms is necessary when searching for minimal models with respect to the number of elements. However, flattening can also be detrimental for performance of reasoning methods due to weakening the role of unification. In this paper we propose to reduce the amount of flattening without compromising completeness with respect to finite satisfiability. Our main idea is to use cyclic/non-cyclic sort separation to restrict flattening to a subset of sorts, which we call the sort-restricted flattening. In this way we avoid flattening of certain terms and translate the problem of finite model finding into a sequence of satisfiability problems in the non-cyclic fragment. Our sort-restricted transformation is complete for finite satisfiability but the minimality requirement on the obtained models is relaxed: only flattened sorts will be minimal. Since the non-cyclic fragment can be decided by instantiation-based methods it is natural to to use instantiation based reasoning systems such as iProver [19], Darwin [4] or Equinox [8].

We implemented our sort classification algorithm in iProver and evaluated it over the TPTP library [28] which is the largest available collection of first-order problems. Since most of the problems in TPTP are unsorted we use a sort inference algorithm to automatically annotate first-order problems with sorts, which is similar to one used by Paradox [10]. Our experiments show interesting results. We observe that many problems in domains ranging from verification to knowledge representation contain combinations of EPR, non-cyclic and cyclic sorts. Therefore it seems promising to advance reasoning methods which can benefit from a such combination. Second, we show that using our sort-restricted transformation it was possible to solve a large class of problems with the rating 1 from TPTP v5.3 (54 in total), which could not be solved by any other known system. The sort-restricted transformation helped iProver to win the FNT (First-order Non-Theorems) division at CASC@Turing 2012.

Related work. In [3], an EPR-based translation of finite satisfiability was developed based on exhaustive flattening of terms. In this paper we provide a translation into the non-cyclic fragment of many-sorted logic which allows one to avoid unnecessary flattening. Many-sorted logic has been intensively used in the SMT community and its advantage for first-order modelling has been advocated in e.g., [1, 12, 14]. The non-cyclic fragment can be shown to be equivalent to the St_0 fragment in [1]. In this paper we present a linear-time algorithm for checking whether the formula is in the non-cyclic fragment, propose a separation of sorts into cyclic and non-cyclic with applications to finite satisfiability, and argue that instantiation-based methods are decision procedures for this fragment. An interesting method for satisfiability of quantified formulas in the SMT setting was presented in [17], which is based on representing relevant domains using set constraints. This approach can be also applied to first-order logic, but unlike our method, is not necessarily complete with respect to finite satisfiability. It would be interesting to investigate how our method can be combined with [17], and a very recent approach to finite model finding in the SMT setting presented in [24, 25].

2 Preliminaries

In this paper we consider many-sorted first-order logic with equality. A signature is a tuple $\Sigma = \langle \mathcal{S}, \mathcal{F}, \mathcal{P}, \mathit{arity}_{\mathcal{F}}, \mathit{arity}_{\mathcal{P}} \rangle$ consisting of a non-empty set of sorts $\mathcal{S}$, a set of function symbols $\mathcal{F}$, a set of predicate symbols $\mathcal{P}$, arity functions $\mathit{arity}_{\mathcal{F}} : \mathcal{F} \to \mathcal{S}^* \times \mathcal{S}$ and $\mathit{arity}_{\mathcal{P}} : \mathcal{P} \to \mathcal{S}^*$, where $\mathcal{S}^*$ denotes the set of finite sequences of sorts. For a function symbol f with arity $\mathit{arity}_{\mathcal{F}}(f) = \langle\langle s_0, \ldots, s_{n-1}\rangle, s_n\rangle$, we call $s_0, \ldots, s_{n-1}$ *argument sorts* and s_n the *value sort* of f. A constant is a function with the empty sequence of arguments. We assume that there is at least one constant of each sort. In this paper we consider only finite signatures. For each sort s we consider a countable set of variables of this sort denoted as V_s. Equality over a sort s will be denoted by $\simeq_s$ and is assumed to be a logical symbol (not included in Σ), defining equality over elements of the same sort. We will omit index s in $\simeq_s$ when the sort is clear from the context. Well-sorted terms and atoms are built from variables and sort-respecting applications of function and predicate symbols in the usual way. We say that a term t is of sort s, denoted as $\mathit{sort}(t) = s$, if either t is a variable of sort s or the top function symbol of t has the value sort s. A literal is an atom or its negation and a clause is a multi-set of literals. We will not distinguish clauses equivalent up to renaming of variables.

A *many-sorted interpretation (structure)* A (over Σ) consists of 1) a domain $\mathrm{dom}(A)$ which is a disjoint union of non-empty sets $\cup_{s\in\mathcal{S}} A_s$ indexed by sorts, 2) a collection of functions $f^A : A_{s_0} \times \cdots \times A_{s_{n-1}} \mapsto A_{s_n}$ where $\mathit{arity}_{\mathcal{F}}(f) = \langle\langle s_0, \ldots, s_{n-1}\rangle, s_n\rangle$, for each $f \in \mathcal{F}$, and 3) a collection of relations $P^A : A_{s_0} \times \cdots \times A_{s_{n-1}}$ where $\mathit{arity}_{\mathcal{P}}(P) = \langle s_0, \ldots, s_{n-1}\rangle$ for each $P \in \mathcal{P}$.

Unsorted first-order logic can be seen as an instance of sorted logic with a single sort.

An expression is ground if it does not contain variables. A *Herbrand universe* over a signature Σ is the set of all ground terms. It is folklore knowledge that automated reasoning methods such as resolution, superposition and instantiation can be straightforwardly adapted from unsorted logic to many-sorted by changing the unification algorithm to sort-aware unification.

3 Non-cyclic Sorts and Finite Herbrand Universe

First we consider the EPR fragment of first-order logic in clausal form, which can be defined as follows. An *EPR signature* is a finite signature which does not contain function symbols other than constants. The *EPR fragment in clausal form* consists of sets of clauses over an EPR signature. One of the main properties of EPR signatures is that the set of all ground terms (the *Herbrand universe*) over such a signature is finite. This implies that the set of all (not necessarily ground) instances of any finite set of EPR clauses is also finite. A direct consequence of this is that reasoning methods such as Inst-Gen [15, 20], Model Evolution [6], Equinox [8], BUMG [5] and DPLL(SX) [23] are decision procedures for the EPR fragment.

In unsorted first-order logic EPR signatures are exactly the signatures with a finite Herbrand universe. As noted in [1, 14], in the presence of sorts, signatures different from EPR can also have a finite Herbrand universe. We characterise them below.

Definition 1. *Consider a signature $\Sigma = \langle \mathcal{S}, \mathcal{F}, \mathcal{P}, arity_{\mathcal{F}}, arity_{\mathcal{P}} \rangle$. A* sort dependency graph *of Σ is a directed graph $SD(\Sigma) = \langle \mathcal{S}, SR \rangle$ with the set of vertices $\mathcal{S}$ and the edge relation SR such that $SR(s_1, s_2)$ if and only if there is a function symbol $f \in \mathcal{F}$ with an argument sort s_1 and the value sort s_2.*

A *path* (of *length* n) in a graph G is a sequence of vertices $v_0, \ldots, v_n$ such that each pair (v_i, v_{i+1}) is in the edge relation of G, where $0 \leq n$ and $0 \leq i < n$. Note that we allow a path to be of the zero length, i.e., consist of a single vertex. A path is *non-trivial* if its length is strictly greater than 0.

Definition 2. *A sort s is called* cyclic *in Σ if there exists a non-trivial path in the sort dependency graph from s to s, otherwise it is called non-cyclic. A signature Σ is called* cyclic *if there is a cyclic sort in Σ and otherwise it is called non-cyclic.*

Non-cyclic signatures can be seen as a natural generalisation of the EPR signatures preserving the property of having a finite Herbrand universe.

Proposition 1. *The Herbrand universe over any non-cyclic signature is finite. Conversely, if a Herbrand universe over a signature is finite then the signature is non-cyclic.*

Proof. It is easy to see that the depth of any term is bounded from above by the longest path in the sort dependency graph which in the case of non-cyclic signatures is bounded by the number of function symbols. In the case when a signature is cyclic we can construct terms of unbounded depth. From this proposition follows.

We define *the non-cyclic clausal fragment of first-order logic* to consist of sets of clauses over a non-cyclic signature. In a similar way as for the EPR fragment it is easy to see that instance based methods are also decision procedures for the non-cyclic fragment. We formulate this as a theorem for Inst-Gen [15,20] and Inst-Gen-Eq [16,22] but it also holds for other instantiation based methods such as Model Evolution. Inst-Gen is an instantiation-based method, complete for first-order logic and Inst-Gen-Eq is its extension with superposition-based equational reasoning. In a nutshell, Inst-Gen and Inst-Gen-Eq combine efficient ground reasoning with gradual instantiations of clauses, based on first-order reasoning.

Theorem 1. *Inst-Gen and Inst-Gen-Eq are decisions procedures for the non-cyclic fragment with equality.*

Proof. (Sketch) Consider a set of clauses over a non-cyclic signature Σ. From Proposition 1 it follows that the Herbrand universe over Σ is finite. Therefore the maximal depth of terms (including non-ground terms) is also bounded. The Inst-Gen calculus only generates instances of the original clauses and since the depth of terms is bounded there are only finitely many such instances. The Inst-Gen calculus treats equality axiomatically. Let us note that adding axioms of equality does not change the non-cyclicity of a clause set.

Inst-Gen-Eq replaces axiomatic equality with superposition-based equational reasoning. Inst-Gen-Eq uses substitutions extracted from unit superposition proofs for instantiating the original clauses. Since the term depth is bounded, there are only finitely many such superposition proofs and only finitely many instances of clauses can be generated.

After analysing problems from the TPTP library we observed that in many cases problems do not fall completely into the EPR or more generally non-cyclic fragment but rather contain combinations of EPR/non-cyclic sorts with cyclic sorts. We refer to Section 7 for details. Our next goal is to partition the signature into cyclic and non-cyclic parts and extend finite model finding methods to gain from this partition.

A *strongly connected component (SCC)* of a directed graph G is a maximal induced subgraph G' of G such that for each pair of vertices in G' there is a path connecting them in G'. A vertex v in a graph G is called *looping* if there is an edge from v to v. An SCC of a graph is called *trivial* if it consists of a non-looping vertex, otherwise it is called *non-trivial.*

Proposition 2. *Consider a signature Σ. A sort s is cyclic in Σ if and only if s belongs to a non-trivial SCC of the sort dependency graph of Σ.*

Proof. If a sort s belongs to a non-trivial SCC then either: 1) s is looping and hence s is cyclic, or 2) there is another sort s' in the same SCC. In the latter case there is a path from s to s' and a path from s' to s and therefore s is also cyclic. For the other direction it is easy to see that if there is a non-trivial path from s to s then this path belongs to the same SCC and therefore this SCC is non-trivial.

The set of all SCCs of the sort dependency graph of a signature Σ will be denoted by $SCC(\Sigma)$. Define the set of cyclic sorts over Σ as $\mathcal{CS}(\Sigma)$ and the set of non-cyclic sorts as $\mathcal{NCS}(\Sigma)$. For any signature Σ, the set of sorts of this signature $\mathcal{S}$ can be partitioned into cyclic and non-cyclic sorts, i.e., $\mathcal{S} = \mathcal{CS}(\Sigma) \cup \mathcal{NCS}(\Sigma)$.

Next we use Tarjan's linear-time algorithm for finding strongly connected components of directed graphs [29] for partitioning a signature into cyclic and non-cyclic sorts.

Theorem 2. *There is a linear-time algorithm that given a signature Σ partitions sorts of Σ into cyclic and non-cyclic sorts.*

Proof. The required algorithm can proceed as follows.

1. Construct the sort dependency graph $SD(\Sigma)$ from Σ.
2. Apply Tarjan's linear time algorithm [29] to obtain the set of all strongly connected components $SCC(\Sigma)$ of $SD(\Sigma)$.
3. The set of all sorts that occur in trivial components of $SCC(\Sigma)$ will be the set of all non-cyclic sorts $\mathcal{NCS}(\Sigma)$ and the set of all sorts that occur in non-trivial components is the set of all cyclic sorts $\mathcal{CS}(\Sigma)$.

It is easy to see that all three steps can be done in time linear in the size of the signature.

Theorem 2 also provides us with a linear time algorithm for checking whether a given signature is cyclic: partition the sort dependency graph into SCC and check whether there is a non-trivial component.

Example 1. Consider a signature $\Sigma = \langle \mathcal{S}, \mathcal{F}, \mathcal{P}, arity_{\mathcal{F}}, arity_{\mathcal{P}} \rangle$ where $\mathcal{S} = \{s_0, s_1, s_2\}$, $\mathcal{F} = \{f, g, h, c_0, c_1, c_2\}$ and $arity(f) = \langle \langle s_0, s_1 \rangle, s_2 \rangle$, $arity(g) = \langle \langle s_0 \rangle, s_0 \rangle$, $arity(h) = \langle \langle s_0 \rangle, s_1 \rangle$ and $arity(c_i) = \langle \langle \rangle, s_i \rangle$ for $0 \leq i \leq 2$. An example of a well-formed term in this signature can be $t = f(g(x), h(g(g(x))))$, where x is a variable of sort s_0. The sort dependency graph $SD(\Sigma)$ and its strongly connected components $SCC(\Sigma)$ are shown on Figure 1. From this we can see that the set of cyclic sorts is $\mathcal{CS}(\Sigma) = \{s_0\}$ and the set of non-cyclic sorts is $\mathcal{NCS}(\Sigma) = \{s_1, s_2\}$. We can see that the term t is of a non-cyclic sort but contains subterms of cyclic and non-cyclic sorts.

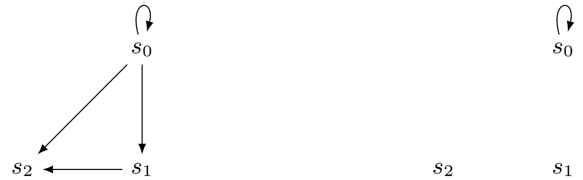


Fig. 1. The sort dependency graph $SD(\Sigma)$ and its strongly connected components $SCC(\Sigma)$

4 EPR-Based Finite Model Finding

In this section we overview a translation of the finite satisfiability problem into the EPR fragment on which we will base our translation into the non-cyclic fragment. Our presentation follows [3] with a slight adaptation to the sorted setting. Let us recapture several notions from [3]. An atom is called *(completely) flat* if it has one of the following forms: 1) $p(x_0, \ldots, x_{n-1})$ where p is a predicate, 2) $x \not\simeq f(x_0, \ldots, x_{n-1})$, where f is a function symbol, or 3) $x \simeq y$. Let us note that function symbols can occur only in flat atoms of the from $x \not\simeq f(x_0, \ldots, x_{n-1})$. A literal is flat if its atom is flat. A clause is flat if all its literals are flat. Consider a clause $C[t]$. A *flattening transformation* applied to $C[t]$ wrt. t produces a clause $x \not\simeq t \vee C[x]$ where x is a fresh variable. By applying the flattening transformation we can transform any set of clauses into an equivalent set of flat clauses [2,3,7]. Let $\mathcal{FT}(S)$ denote a set of flat clauses obtained from S by applying the flattening transformation.

Consider a set of flat clauses S over a signature Σ. For each function symbol in Σ with arity $arity(f) = \langle \langle s_0, \ldots, s_{n-1} \rangle, s_n \rangle$ we introduce a new predicate symbol P_f with arity $arity(P_f) = \langle s_0, \ldots, s_{n-1}, s_n \rangle$. Informally, the predicate P_f will be used to represent the function f with the last argument representing the value of f. We call these introduced predicates as *function predicates*. Now we can eliminate functions from our clause set by applying the following *function elimination transformation*:

$$y \not\simeq f(x_0, \ldots, x_{n-1}) \vee C \Rightarrow \neg P_f(x_0, \ldots, x_{n-1}, y) \vee C.$$

Let $\mathcal{FE}(S)$ denote the set of clauses obtained by exhaustive applications of function elimination to S. In order to ensure that a function predicate P_f represents a graph of a function we need to require P_f to be left-total and right-unique, as defined below. Consider an interpretation I with a relation P of $arity(P) = \langle s_0, \ldots, s_{n-1}, s_n \rangle$, where $0 \leq n$. The relation P is called *left-total* in I if for every sequence of elements $a_0, \ldots, a_{n-1}$ in I, of the respective sorts $s_0, \ldots, s_{n-1}$, there exists an element $a_n \in I$ of sort s_n such that $P(a_0, \ldots, a_n)$ holds in I. The relation P is called *right-unique* in I if whenever $I \models P(a_0, \ldots, a_{n-1}, a_n)$ and $I \models P(a_0, \ldots, a_{n-1}, a'_n)$ then $I \models a_n \simeq a'_n$. It is shown in [3] that we can drop the right-uniqueness requirement when we consider flat clauses, preserving finite satisfiability. When we consider finite domains we can express left-totality in the EPR fragment as shown below.

Let us consider finite satisfiability of flat clauses. For each sort s we fix a finite set of constants $d^s_1, \ldots, d^s_k$, called *domain constants*, representing all elements of this sort. Collection of all domain constants will be called a *constant domain* and denoted by $\mathcal{D}$.

Then left-totality of a function predicate P_f over a constant domain can be expressed using the following *totality axiom*:

$$P_f(x_0, \ldots, x_{n-1}, d^s_1) \vee \cdots \vee P_f(x_0, \ldots, x_{n-1}, d^s_k),$$

where $d^s_1, \ldots, d^s_k$ are all domain constants of the sort s.

For a constant domain $\mathcal{D}$, let $TAx(\mathcal{D})$ denote the set of all totality axioms of function predicates in the signature. For a set of clauses S and a constant domain $\mathcal{D}$ we call the set of clauses $BFM(S, \mathcal{D}) = \mathcal{FE}(\mathcal{FT}(S)) \cup TAx(\mathcal{D})$ the *basic finite model finding translation* of S with respect to $\mathcal{D}$.

Theorem 3. *[3] A set of clauses S is satisfiable over a finite constant domain $\mathcal{D}$ if and only if $BFM(S, \mathcal{D})$ is satisfiable.*

As shown in [3], due to flattening it is also possible to eliminate the equality predicate altogether by introducing axioms stating disequality of the domain constants: $\mathcal{E}(\mathcal{D}) = \{d^s_i \not\simeq_s d^s_j \mid i \neq j, d^s_i, d^s_j \in \mathcal{D}\}$. After adding the disequality axioms one can replace the equality predicate $\simeq_s$ by a fresh binary predicate $\mathcal{E}_s$ for each sort s, preserving satisfiability. We denote this translation as $BFME(S, \mathcal{D})$.

Let us note that the result of applying any of the translations BFM, or $BFME$ is always an EPR set of clauses, and therefore instantiation-based methods can be used for checking satisfiability of $BFM(S, \mathcal{D})$ and $BFME(S, \mathcal{D})$.

Without loss of generality we can restrict ourselves to the Herbrand interpretations which in this case are built over the domain constants. The search for finite satisfiability then starts with a constant domain consisting of a single constant in each sort and then proceeds by iteratively adding new constants until $BFME(S, \mathcal{D})$ becomes satisfiable. One of the properties of this approach is that if a set of clauses is finitely satisfiable then we obtain a minimal model with respect to the number of domain elements.

5 Flattening and Finite Model Finding

As we have seen, flattening is essential for the basic finite model finding translation. Unfortunately, it also can have a detrimental effect on reasoning methods.

Example 2. Consider an unsorted signature consisting of n unary predicates $P_1, \ldots P_n$ and n constants $c_1, \ldots, c_n$. Consider the following set of ground unit clauses:

$$S = \bigcup_{1 \leq i \leq n} \{P_i(c_i)\} \bigcup \bigcup_{1 \leq i < j \leq n} \{\neg P_i(c_j)\}.$$

Satisfiability of S can be trivially shown by propositional reasoning, (considering $P_i(c_j)$ as propositional atoms). Let us consider the basic finite model translation applied to S. After flattening and introduction of function predicates P_{c_i} for each constant c_i, $1 \leq i \leq n$ we obtain the following set of clauses:

$$\mathcal{FE}(\mathcal{FT}(S)) = \bigcup_{1 \leq i \leq n} \{\neg P_{c_i}(x) \vee P_i(x)\} \bigcup \bigcup_{1 \leq i < j \leq n} \{\neg P_{c_j}(x) \vee \neg P_i(x)\}.$$

It is easy to see that any satisfying interpretation for this set of clauses will have the domain size at least n. Therefore the finite model finding procedure will iteratively increase the domain size, by adding domain axioms until n is reached. Moreover reasoning with intermediate domain sizes can be nontrivial due to introduced symmetries.

Example 2 is deliberately simple. Let us slightly modify this example by adding a "dummy" argument to each predicate P_i and replace $P_i(c_j)$ with $P_i(c_j, f(x))$. Then, although this change obviously does not affect satisfiability, the resulting set is challenging for instantiation-based systems.

We can see that even for simple problems flattening can introduce variables into the problem and increase the search space. Our approach aims at reducing the amount of unnecessary flattening. First we can observe that if our problem is EPR as in Example 2, then we can apply instantiation-based methods directly to such a problem without applying the flattening transformation. In the next section we restrict flattening further to terms of cyclic sorts.

6 Sort-Restricted Flattening

Our approach is to restrict flattening to specified sorts and at the same time keep the resulting translation in the non-cyclic fragment.

Consider a signature Σ with the set of sorts $\mathcal{S}$. Consider a subset of $\mathcal{S}' \subseteq \mathcal{S}$. We say that an interpretation I is *$\mathcal{S}'$-finite* if each sort in $\mathcal{S}'$ has a finite domain in I. A formula is *$\mathcal{S}'$-finitely satisfiable* if it is satisfied in an $\mathcal{S}'$-finite interpretation. $\mathcal{S}'$-finite satisfiability generalises finite satisfiability, that is if a formula is finitely satisfiable then it is also $\mathcal{S}'$-finitely satisfiable for any $\mathcal{S}' \subseteq \mathcal{S}$, but the converse in general does not hold.

The *sort-restricted flattening transformation* with respect to $\mathcal{S}'$ is defined as follows:

$$L[t] \vee C \Rightarrow x \not\simeq t \vee L[x] \vee C,$$

where:

1. t is not a variable,
2. $sort(t) \in \mathcal{S}'$,
3. $L[t]$ has one of the forms: $t \simeq s$, $s \simeq t$, $t \not\simeq s$, or $(\neg)P[t]$ for a predicate P, and
4. x does not occur in $L[t] \vee C$.

The result of exhaustive application of the sort-restricted flattening transformation to a set of clauses S, is denoted as $\mathcal{FTR}(\mathcal{S}', S)$.

We sort-restrict other ingredients of the finite satisfiability transformation:

1. function elimination is restricted to functions with value sorts in $\mathcal{S}'$, denoted by $\mathcal{FER}(\mathcal{S}', S)$;
2. a sort-restricted finite constant domain (or just sort-restricted constant domain) is a collection of constants $\bar{d}^{s_1}, \ldots, \bar{d}^{s_m}$, denoted by $\mathcal{DR}(\mathcal{S}')$, where $\mathcal{S}' = \{s_1, \ldots, s_m\}$ and each $\bar{d}^{s_i}$ is a non-empty sequence of constants of sort s_i. We assume that domain constants are fresh for the signature Σ;
3. totality axioms for function predicates over a sort-restricted constant domain $\mathcal{DR}(\mathcal{S}')$ are defined as in Section 4 and will be denoted as $TAx(\mathcal{DR})$.

Consider a set of clauses S over signature Σ. We say that S is satisfiable in a sort-restricted constant domain $\mathcal{DR}(\mathcal{S}')$ (or $\mathcal{DR}(\mathcal{S}')$ *satisfiable*) if there is a model I of S which can be expanded with constants from $\mathcal{DR}(\mathcal{S}')$ so that each element in I of a sort $s \in \mathcal{S}'$ is named by a constant in $\mathcal{DR}(\mathcal{S}')$. It is easy to see that S is $\mathcal{S}'$*-finitely* satisfiable if and only if there is a sort-restricted constant domain $\mathcal{DR}(\mathcal{S}')$ such that S is $\mathcal{DR}(\mathcal{S}')$ satisfiable.

For a set of clauses S, a subset of sorts $\mathcal{S}' \subseteq \mathcal{S}$ and a sort-restricted constant domain $\mathcal{DR}(\mathcal{S}')$ we call the set of clauses $BFMR(\mathcal{S}', S, \mathcal{DR}) = \mathcal{FER}(\mathcal{S}', \mathcal{FTR}(\mathcal{S}', S)) \cup TAx(\mathcal{DR})$ *the sort-restricted finite model finding translation* (or just the sort-restricted translation) of S with respect to $\mathcal{S}'$ and $\mathcal{DR}(\mathcal{S}')$.

Theorem 4. *Consider a signature Σ with a set of sorts $\mathcal{S}$, a subset of sorts $\mathcal{S}' \subseteq \mathcal{S}$ and a sort-restricted constant domain $\mathcal{DR}(\mathcal{S}')$. A set of clauses S over Σ is $\mathcal{DR}(\mathcal{S}')$ satisfiable if and only if the sort-restricted translation $BFMR(\mathcal{S}', S, \mathcal{DR})$ is satisfiable.*

Proof. Adaptation of results from [3].

Similar to the basic case we can eliminate equality predicate over sorts in $\mathcal{S}'$ by first adding axioms stating disequality of the domain constants:

$$\mathcal{E}(\mathcal{DR}) = \{d_i^s \not\simeq_s d_j^s \mid i \neq j, d_i^s, d_j^s \in \mathcal{DR}\}$$

and then replacing $\simeq_s$ by a fresh binary predicate $\mathcal{E}_s$ for each sort $s \in \mathcal{S}'$. We denote this translation as $BFMER(\mathcal{S}', S, \mathcal{DR})$. Let us note that after applying $BFMER(\mathcal{S}', S, \mathcal{DR})$, equality can still remain between terms of sorts which are not in $\mathcal{S}'$.

In order to keep $BFMER(\mathcal{S}', S, \mathcal{DR})$ in a decidable fragment we propose to restrict flattening to a superset of cyclic sorts. For this, we define a set of sorts to which we do not apply flattening to be any subset of non-cyclic sorts $\mathcal{NCS}(\Sigma)$, which we call *non-flattening sorts* and denote by $\mathcal{NFS}(\Sigma)$. Then, the set of sorts to which we apply flattening will be $\mathcal{S} \setminus \mathcal{NFS}(\Sigma)$, which we call *flattening sorts* and denote by $\mathcal{FS}(\Sigma)$. Examples of non-flattening sorts include the empty set of sorts, the set of EPR sorts and the set of all non-cyclic sorts.

Proposition 3. *Consider a signature Σ, a set of flattening sorts $\mathcal{FS}(\Sigma)$ and a sort-restricted constant domain $\mathcal{DR}(\mathcal{FS})$. Then, for any set of clauses S the sort-restricted translation $BFMER(\mathcal{FS}, S, \mathcal{DR})$ is in the non-cyclic fragment.*

The search for finite satisfiability then starts with a sort-restricted constant domain consisting of a single constant in each flattening sort and then proceeds by iteratively adding new constants into the sort-restricted domain until $BFMER(\mathcal{FS}, S, \mathcal{D})$ becomes satisfiable. From Theorem 4 and our remarks above it follows that this method is complete with respect to finite model finding and more generally with respect to $\mathcal{FS}(\Sigma)$-finite model finding. In particular, if a set of clauses has a finite model then the procedure will find a sort-restricted constant domain which satisfies this set of clauses.

Let us note an essential difference between basic and sort-restricted translations: in the basic case the finite model finding is restricted to minimal models (with respect to the number of elements), whereas in the sort-restricted case this requirement is relaxed so that only domains of flattening sorts will be minimal but domains of non-flattening sorts can be arbitrary interpretations.

Example 3. Consider the problem from Example 2. Since all sorts in this example are EPR we can take them as the set of non-flattening sorts $\mathcal{NFS}(\Sigma)$. In this case the sort-restricted finite model finding transformation will not change the set of clauses.

Example 4. Let us consider the signature Σ from Example 1 and a clause C:

$$f(x, h(g(x))) \simeq f(x, c_1) \vee h(x) \simeq c_1.$$

Let us apply sort-restricted finite model finding transformation to C with the set of non-flattening sorts to be the set of all non-cyclic sorts: $\mathcal{NFS} = \{s_1, s_2\}$. The result of exhaustive application of sort-restricted flattening to C is:

$$y \not\simeq g(x) \vee f(x, h(y)) \simeq f(x, c_1) \vee h(x) \simeq c_1.$$

The result of sort-restricted function elimination is:

$$\neg P_g(x, y) \vee f(x, h(y)) \simeq f(x, c_1) \vee h(x) \simeq c_1.$$

And the domain axioms are of the form:

$$P_g(x, d_1^{s_0}) \vee \ldots \vee P_g(x, d_k^{s_0}),$$

where $d_1^{s_0}, \ldots, d_k^{s_0}$ are domain constants of sort s_0.

Let us compare this to the case when $\mathcal{NFS}$ is the empty set, which reduces the sort-restricted transformation to the basic transformation.

After applying flattening to C we obtain a much longer clause:

$$\begin{gathered} y_1 \not\simeq g(x) \vee y_2 \not\simeq h(y_1) \vee y_3 \not\simeq c_1 \vee y_4 \not\simeq f(x, y_2) \vee y_5 \not\simeq f(x, y_3) \vee y_6 \not\simeq h(x) \\ \vee \\ y_4 \simeq y_5 \vee y_6 \simeq y_3. \end{gathered}$$

We can also note that in this example basic flattening introduces positive equations between variables like $y_4 \simeq y_5$ and $y_6 \simeq y_3$ which can be problematic for reasoning methods.

After function elimination we obtain:

$$\begin{gathered} \neg P_g(x, y_1) \vee \neg P_h(y_1, y_2) \vee \neg P_{c_1}(y_3) \vee \neg P_f(x, y_2, y_4) \vee \neg P_f(x, y_3, y_5) \vee \neg P_h(x, y_6) \\ \vee \\ y_4 \simeq y_5 \vee y_6 \simeq y_3. \end{gathered}$$

And domain axioms are of the form:

$$\begin{gathered}
P_g(x_0, d_1^{s_0}) \vee \ldots \vee P_g(x_0, d_{k_0}^{s_0}) \\
P_h(x_0, d_1^{s_1}) \vee \ldots \vee P_h(x_0, d_{k_1}^{s_1}) \\
P_{c_1}(d_1^{s_1}) \vee \ldots \vee P_{c_1}(d_{k_1}^{s_1}) \\
P_f(x_0, x_1, d_1^{s_2}) \vee \ldots \vee P_f(x_0, x_1, d_{k_2}^{s_2}).
\end{gathered}$$

As we can see from this example, basic transformation can result in a considerably larger set of clauses. Moreover positive equations between variables can be introduced which can be avoided when the sort-restriction transformation is applied.

Iterative flattening. Let us briefly discuss an extension of our method which further refines flattening applications. This is based on the following observation. Consider a signature Σ with the set of sorts $\mathcal{S}$. If we apply the sort-restricted flattening to a single sort $s \in \mathcal{S}$, then all function symbols with the value sort s will be replaced by predicate symbols, resulting in a new signature Σ'. It is easy to see that the sort dependency graph for Σ' can be obtained from the sort dependency graph of Σ by removing edges adjacent to s. In particular, if we pick s from a non-trivial strongly connected component, after eliminating all edges adjacent to s we may be able to decompose this strongly connected component further into smaller components. The process of flattening sorts one by one and decomposing the corresponding strongly connected components can be repeated until only trivial components remain. The advantage of this approach is that we can reduce the number of sorts that require flattening even further. As an example let as consider a signature Σ with the set of sorts $\mathcal{S} = \{s_0, \ldots, s_n\}$, forming a single cycle $s_0, \ldots, s_n, s_0$ in the sort dependency graph. There is only one strongly connected component in this sort dependency graph, which is this cycle itself. After flattening only one sort, say s_0, the sort dependency graph can be completely decomposed into trivial strongly connected components. This allows us to avoid flattening of the remaining sorts $\{s_1, \ldots, s_n\}$.

7 Implementation and Evaluation

Our implementation is based on the iProver system [19]. iProver is based on the Inst-Gen calculus which is complete for first-order logic. As we argued in Section 3, Inst-Gen is also a decision procedure for the non-cyclic fragment. iProver accepts first-order problems in CNF form. For problems in full first-order syntax (FOF) we used Vampire [18, 26] as an external clausifier, optionally E prover [27] can also be used as a clausifier.

For our evaluation we used the TPTP library v5.3 [28], which contains 15,550 FOF and CNF problems. Our experiments were run on a cluster of Linux machines with memory limit 2GB and 2.33GHz CPU. We separate our experimental results into two classes. The first class is related to sort inference and the second class is related to experiments with sort-restricted finite model finding.

Sort inference. iProver implements a sort inference algorithm similar to one implemented in Paradox [10] which transforms unsorted first-order clauses into a set of sorted clauses. This algorithm first assigns different sorts to all predicate arguments, function arguments and values, then it applies a union-find algorithm to merge sorts forced to be equal due to variable dependencies, or occurrences of the equality predicate. Extracted sorts are monotone in the sense of [9], which means that for any model satisfying a set of clauses we can extend domain of any sort with new elements without affecting satisfiability. The overall sort inference resulted in 4,090 problems with non-trivial sorts. We implemented an algorithm for classifying sorts into cyclic, EPR and non-cyclic sorts as described in Section 3. Our implementation uses an OCaml library ocamlgraph [11] for computing strongly connected components of directed graphs.

There are 1,383 problems which were recognised by iProver as being pure EPR problems (after clausification). For clarity we exclude pure EPR problems from experiments below. Of course, all discussed methods are trivially applicable to them. We found that after removing pure EPR problems, 1,195 remaining problems have at least one non-cyclic sort, which is around 1/3 of all problems with non-trivial sorts; 1,077 problems have at least one EPR sort. Collectively over all problems there are 56,679 sorts, 18,502 non-cyclic sorts and among them 9,569 EPR sorts . From this we can see that the number of EPR sorts is approximately the same as non-EPR non-cyclic sorts. Also, we can see that most problems with non-cyclic sorts combine EPR and non-EPR non-cyclic sorts.

Problems with non-cyclic sorts are spreading over many domains of TPTP (even after excluding pure EPR problems), most notably software and hardware verification: SWV, SWW, HWV; knowledge representation SWB, KRS, CSR; algebra: TOP, GEO, SET, SEU, RNG; natural language processing: NLP; planning: PLA; and other domains: PUZ, MGT, MSC, SYN. This indicates that non-cyclic sorts occur naturally in many applications and we believe reasoning methods can be tuned to benefit from this. We also believe that if problems were sorted by domain experts rather than by using automatic sort inference, considerably more problems would be identified to have sorts and non-cyclic sorts in particular.

Sort-restricted finite model finding. We implemented our sort-restricted finite model finding translation *BFMER*, as described in Section 6. Our implementation also features symmetry breaking based on sorts similar as it is done in Paradox. Using sort-restricted finite model finding we were able to solve 54 problems in TPTP v5.3 with the rating 1, all from the KRS domain. We also found a bug in TPTP v5.3 where a problem with the rating 1 (KRS264+1), was stated to be a "Theorem" but our experiments showed it to be satisfiable. We thank Geoff Sutcliffe for helping to debug this problem which resulted in fixing an axiomatisation in the most recent version of TPTP v5.4. iProver with the sort-restricted finite model finding participated in the latest CASC competition, and these enhancements helped iProver to win the FNT (First-order Non-Theorems) division at CASC@Turing 2012. iProver also participated in the evaluation of TPTP v5.4. As the result, some satisfiable problems with the rating 1 in TPTP v5.3 are of a lower rating in TPTP v5.4. In total, iProver solved 72% of problems which are classified as Satisfiable or CounterSatisfiable in TPTP v5.3. We experimented with two cases of non-flattening sorts: 1) all EPR sorts, and 2) all non-cyclic sorts. We observed that most problems are solved in the first case (72%). In the second case, there are a number of problems which could not be solved by the first case, but overall performance is a bit worse: only 69% of problems were solved. One possible explanation can be that flattening can still be beneficial in some cases due to axiomatic treatment of equality in iProver. We expect that this can be amended by using iProver-Eq [21] which integrates equality using superposition-based reasoning. Another explanation can be that in some cases searching for minimal models can still be quicker. Our method gives flexibility on which sorts to flatten, we can choose any superset of cyclic sorts or apply even more fine-grained flattening based on iterative flattening as discussed in Section 6. We leave it for the future work to find best strategies for selecting sorts for flattening.

8 Conclusion and Future Work

In this paper we investigated the non-cyclic fragment of many-sorted first-order logic with equality. We showed that the non-cyclic fragment is decidable by instantiation-based methods. We presented a linear time algorithm for checking whether a given signature is non-cyclic and more generally for classifying sorts into non-cyclic, EPR and cyclic. We presented a translation of finite model finding into a sequence of satisfiability problems in the non-cyclic fragment, which avoids flattening terms of non-cyclic sorts. We implemented our sort classification and finite model finding translation in iProver. Experimental results are encouraging and we were able to solve a large class of problems which could not be solved by other systems.

For the future work we are planning to integrate sort-restricted finite model finding into iProver-Eq. We will also investigate how reasoning methods can benefit from our sort classification in the refutation setting. It is interesting to investigate combinations of the non-cyclic fragment with other theories in the spirit of [13]. We are planning to investigated combinations of our approach to finite satisfiability with resent SMT-based approaches [17,24,25].

iProver with implemented features for sort classification and sort-restricted finite model finding is available at: `http://www.cs.man.ac.uk/~korovink/iprover/`

Acknowledgments. The author is grateful to anonymous reviewers for providing detailed comments which helped to improve this paper.

References

1. Abadi, A., Rabinovich, A., Sagiv, M.: Decidable fragments of many-sorted logic. J. Symb. Comput. 45(2), 153–172 (2010)
2. Bachmair, L., Ganzinger, H., Voronkov, A.: Elimination of equality via transformation with ordering constraints. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS (LNAI), vol. 1421, pp. 175–190. Springer, Heidelberg (1998)
3. Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. J. Applied Logic 7(1), 58–74 (2009)
4. Baumgartner, P., Fuchs, A., Tinelli, C.: Implementing the model evolution calculus. International Journal on Artificial Intelligence Tools 15(1), 21–52 (2006)

5. Baumgartner, P., Schmidt, R.A.: Blocking and other enhancements for bottom-up model generation methods. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 125–139. Springer, Heidelberg (2006)
6. Baumgartner, P., Tinelli, C.: The model evolution calculus. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 350–364. Springer, Heidelberg (2003)
7. Brand, D.: Proving theorems with the modification method. SIAM J. Comput. 4(4), 412–430 (1975)
8. Claessen, K.: The anatomy of Equinox - an extensible automated reasoning tool for first-order logic and beyond - (talk abstract). In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 1–3. Springer, Heidelberg (2011)
9. Claessen, K., Lillieström, A., Smallbone, N.: Sort it out with monotonicity - translating between many-sorted and unsorted first-order logic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 207–221. Springer, Heidelberg (2011)
10. Claessen, K., Sörensson, N.: New techniques that improve MACE-style model finding. In: Baumgartner, P., Fermüller, C. (eds.) CADE-19 Workshop: Model Computation Principles, Algorithms, Applications, pp. 11–27 (2003)
11. Conchon, S., Filliâtre, J.-C., Signoles, J.: ocamlgraph, `http://ocamlgraph.lri.fr`
12. Fontaine, P.: Techniques for verification of concurrent systems with invariants. PhD thesis, Institut Montefiore, Université de Liège, Belgium (2004)
13. Fontaine, P.: Combinations of theories and the Bernays-Schönfinkel-Ramsey class. In: VERIFY 2007. CEUR Workshop Proceedings. CEUR-WS.org (2007)
14. Fontaine, P., Gribomont, E.P.: Decidability of invariant validation for paramaterized systems. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 97–112. Springer, Heidelberg (2003)
15. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. In: Proc. 18th IEEE Symposium on LICS, pp. 55–64. IEEE (2003)
16. Ganzinger, H., Korovin, K.: Integrating equational reasoning into instantiation-based theorem proving. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 71–84. Springer, Heidelberg (2004)
17. 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)
18. Hoder, K., Khasidashvili, Z., Korovin, K., Voronkov, A.: Preprocessing techniques for first-order clausification. In: Cabodi, G., Singh, S. (eds.) Formal Methods in Computer-Aided Design (FMCAD 2012), pp. 44–51. IEEE (2012)
19. 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)
20. Korovin, K.: Inst-Gen - A modular approach to instantiation-based automated reasoning. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics. LNCS, vol. 7797, pp. 239–270. Springer, Heidelberg (2013)
21. Korovin, K., Sticksel, C.: iProver-Eq: An instantiation-based theorem prover with equality. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 196–202. Springer, Heidelberg (2010)
22. Korovin, K., Sticksel, C.: Labelled unit superposition calculi for instantiation-based reasoning. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 459–473. Springer, Heidelberg (2010)
23. Piskac, R., de Moura, L., Bjørner, N.: Deciding effectively propositional logic using DPLL and substitution sets. J. Autom. Reasoning 44(4), 401–424 (2010)
24. Reynolds, A., Tinelli, C., Goel, A., Krstić, S.: Finite model finding in SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 640–655. Springer, Heidelberg (2013)

25. Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M., Barrett, C.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 377–391. Springer, Heidelberg (2013)
26. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Communications 15(2-3), 91–110 (2002)
27. 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)
28. 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)
29. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. 1(2), 146–160 (1972)

Detection of First Order Axiomatic Theories

Guillaume Burel[1] and Simon Cruanes[2]

[1] ÉNSIIE/Cédric, 1 square de la résistance, 91025 Évry cedex, France
guillaume.burel@ensiie.fr
http://www.ensiie.fr/~guillaume.burel/
[2] École polytechnique and INRIA, 23 Avenue d'Italie, 75013 Paris, France
simon.cruanes@inria.fr
https://who.rocq.inria.fr/Simon.Cruanes/

Abstract. Automated theorem provers for first-order logic with equality have become very powerful and useful, thanks to both advanced calculi — such as superposition and its refinements — and mature implementation techniques. Nevertheless, dealing with some axiomatic theories remains a challenge because it gives rise to a search space explosion. Most attempts to deal with this problem have focused on specific theories, like AC (associative commutative symbols) or ACU (AC with neutral element). Even detecting the presence of a theory in a problem is generally solved in an ad-hoc fashion. We present here a generic way of describing and recognizing axiomatic theories in clausal form first-order logic with equality. Subsequently, we show some use cases for it, including a redundancy criterion that can be applied to some equational theories, extending some work that has been done by Avenhaus, Hillenbrand and Löchner.

1 Introduction

Automated theorem proving for first order logic has lead to many successful techniques to tackle problems from a lot of application domains. Among the most prominent techniques lies resolution[10]. Superposition[8] appeared later to handle the difficult issue of equality reasoning, that would otherwise drown most provers in a huge search space.

Many theorem provers for first-order logic with equality contain an ad-hoc engine to recognize instances of Associative Commutative (AC) symbols, composed of the two following axioms:

Associativity: $\forall x \forall y \forall z\ x + (y + z) = (x + y) + z$,
Commutativity: $\forall x \forall y\ x + y = y + x$.

Once the automated prover has recognized that some symbol has the AC property, it can use some technique to deal with it. However, if similar techniques can be applied to other axiomatic theories — theories that can be defined in

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 229–244, 2013.

terms of a finite set of axioms — code would need to be written for those provers to handle each new theory. We propose here a system that can recognize the presence of theories in a generic and incremental way. The system is based on the use of a second theorem prover, based on Datalog[1], that reasons about the properties that the problem exhibits, rather than trying to solve the problem itself. In some limited sense, this is similar to what a human mathematician does: she would try to use equations and hypotheses on the problem itself, but at the same time she would recognize already met patterns and specific structures (for instance, a group structure, a linear field, or an isomorphism to some other part of the mathematics) and use this meta knowledge to apply theorems and lemmas she knows.

We implemented this technique in our experimental theorem prover Zipperposition. Zipperposition is free software, available under the GPL license at `https://www.rocq.inria.fr/deducteam/Zipperposition/index.html`. It is written in OCaml and implements ordered superposition, with lazy reduction to CNF and automatic selection of a precedence. An embedded Datalog engine is used to reason on properties of the problems, including which known theories and axioms are present; both systems interact by exchanging clauses on the one hand, deduced properties on the other. The superposition prover can use the additional information to infer new clauses thanks to *lemmas* or to activate theory-specific *redundancy criteria.*

Then, we expose two possible applications. The first is a powerful lemma that allows theorem provers that deal well with equality to discover that some relations represent the graph of a function, and to replace instances of the relation by equations. For instance, in the TPTP[13] archive, many algebraic problems on groups (or extensions thereof) are encoded using $\mathrm{sum}(X, Y, Z)$ instead of $Z = \mathrm{add}(X, Y)$. This complicates the axiomatization (many more axioms, that are big Horn clauses, etc.) compared to an equational view of the problem. Our lemma, fed to the prover in a simple declarative language as:

```
functional(r) is axiom ~r(X,Y,Z) | ~r(X,Y,Z2) | Z=Z2.
total(r, f) is axiom r(X,Y,f(X,Y)).
lemma r(X,Y,Z) <=> Z = f(X,Y) if functional(r) and total(r, f).
```

allows to recover an equational (boolean) definition from this encoding, which can then be unfolded to simplify clauses.

The second application is the per-theory activation of an equational redundancy criterion. If we know a saturated, ground convergent system of equations for some theory [2], literals that are tautological or absurd in this theory can be removed while retaining completeness. Our framework allows us to know when such a theory occurs in a problem, so we can use the corresponding redundancy criterion.

We first expose some basic definitions and notations, then successively expose techniques for recognizing individual axioms and whole theories. Then, after some examples of how to use knowledge about axiomatic theories, we present some experimental results and conclude.

2 Notations and Definitions

The first step toward recognizing theories, is recognizing instances of individual axioms. More complex algebraic theories, like group theory, involve several symbols. We present here a general framework for representing axiom schemas, axiom instances, and for finding instances of the former among the latter. We start with some basic notations.

A *signature* $\Sigma = (S, V)$ is the combination of a finite set of symbols S (with an arity function arity : $S \to \mathbb{N}$), with a countable set of variables V. *Terms* in a signature $\Sigma = (S, V)$ are defined recursively by $t = X \mid f(t_1, \ldots, t_n)$, where $X \in V$ and $f \in S$, with $\text{arity}(f) = n$. We give a *type* to each term, in a simple-type system with base types ι for individuals and o for propositions; a function type is written $(\tau_1 \times \cdots \times \tau_n) \to \tau$. The statement "$t$ has type τ" is written $t : \tau$. The set of variables of a term, vars(t), is recursively defined as

$$\text{vars}(X) = \{X\}$$

$$\text{vars}(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} \text{vars}(t_i)$$

From now on, f, g, h will be symbols, t, t', t_i will be terms, uppercase letters like X, Y, Z will denote variables and τ will be a type.

A *clause* is a disjunction of literals $l_1 \vee \cdots \vee l_n$, each *literal* being an equation $s = t$, or the negation of an equation $s \neq t$ (s and t must have the same type). A literal of any sign is written $s \doteq t$. Following [11], we represent propositions p by boolean-typed equations $p = \top$, where $\top : o$ is a special constant for truth.

A *substitution* σ is a finite mapping from variables to terms. The result of applying a substitution to a term t is noted $t\sigma$. If σ and θ are substitutions, then σ is *more general* than θ if there exists η such that $\forall t.\ t\theta = t\sigma\eta$. Usual notions for *most general unifiers* and *most general matcher* are employed:

unifier: The most general unifier of two terms t_1 and t_2, if it exists, is the most general substitution σ such that $t_1\sigma = t_2\sigma$. Terms for which such a substitution exists are said to be *unifiable*.

matcher: The most general matcher of two terms t_1 and t_2, if it exists, is the most general substitution σ such that $t_1\sigma = t_2$. Similarly, it is not always defined.

Equality of two terms modulo a theory E is written $t_1 =_E t_2$, short for $E \vdash t_1 = t_2$. We extend the definitions of unification and matching to *unification modulo AC* and *matching modulo AC*, respectively defined by $t_1\sigma =_{AC} t_2\sigma$ and $t_1\sigma =_{AC} t_2$.

Let the *local signature* of a term or clause, noted $\mathrm{ls}(t)$, be defined as follow:

$$\begin{aligned}\mathrm{ls}(X) &= \emptyset \\ \mathrm{ls}(f(t_1,\ldots,t_n)) &= \{f\} \cup \bigcup_{i=1}^{n} \mathrm{ls}(t_i) \\ \mathrm{ls}(s \dot{=} t) &= \mathrm{ls}(s) \cup \mathrm{ls}(t) \\ \mathrm{ls}(l_1 \vee \cdots \vee l_n) &= \bigcup_{i=1}^{n} \mathrm{ls}(l_i)\end{aligned}$$

A few special symbols (disjoint from any signature) will be used:

- A *symbol marker*, $\mathfrak{s}$, used to prefix function symbols;
- A *variable marker*, $\mathfrak{v}$, used to prefix variables.

For the meta-prover, we encode properties of the problem at hand in *higher-order logic*, where the definition of terms is extended with *binders* (here, only λ). Well-formed terms for the meta-prover are defined by $t = X \mid f \mid t\ t \mid \lambda X.t$ where $f \in S \cup \{\mathfrak{s}, \mathfrak{v}\}$, $X \in V$ and t a term. Term application $t\ t$ is curried and left-associative. We call *lambda terms* terms in which some variables are bound by a lambda-abstraction. We will assume that the reader knows about basic simply-typed lambda-calculus, but recall that β-reduction is the rule $(\lambda X.t)\ t' \rightarrow_\beta [t'/X]t$ and we will work modulo alpha-equivalence in order to prevent variable captures. In the rest of the paper, $t \downarrow^\beta$ denotes the normal form of a term t w.r.t. β-reduction, i.e., the unique term t' such that $t \rightarrow_\beta^* t'$ and $\neg\exists t''\ t' \rightarrow t''$ (it always exists because simply-typed lambda calculus is convergent).

3 Detecting Axioms

The first step toward recognizing theories, is recognizing instances of individual axioms of first-order theories. We will see, in the next section, how to recognize full theories. Many theorem provers contain an ad-hoc system to recognize instances of Associative Commutative (AC) symbols, composed of two axioms:

Associativity: $\forall x \forall y \forall z\ x + (y + z) = (x + y) + z$,
Commutativity: $\forall x \forall y\ x + y = y + x$.

More complex algebraic theories, like group theory, involve several symbols. We present here a general framework for representing axiom schemas, axiom instances, and for finding instances of the former among the latter. Recognizing axioms in any signature is a higher order problem, but we are going to use *currying* to stay in a first-order setting. We start with some basic notations.

We call *pattern* a clause c parametrized by $\mathrm{ls}(c)$. This notion of pattern is central in our approach, since it allows us to reason over axioms and theories regardless of the actual signature of the problem (a given axiom or theory might have several distinct instances within the same proof). A pattern p is represented as a higher-order *curried* term $t \equiv \lambda X_1 : \tau_1.\lambda X_2 : \tau_2. \dots \lambda X_n : \tau_n.c$, or more compactly $\Lambda_{i=1}^n X_i : \tau_i.c$. We need to curry the term because we cannot replace a function symbol by a variable in first-order terms. c is the *core* of the pattern, and $s_1, \dots, s_n$ the *input types* or *types* of the pattern. Note that although patterns are higher-order terms (because of the lambda abstractions), we still reason over first-order problems, and any instantiation of a pattern must yield a first-order clause.

Patterns are an extension of the *representative patterns* defined in [4], but are more general because they are curried and deal with non-unit clauses, which explains our use of AC-matching. In addition to that, our technique is concerned with which set of symbols instantiates a given pattern. On the other hand, representative patterns are indexed by an AVL tree, which makes the matching process very efficient.

The point in using curried terms to represent patterns is that we can leverage many well-known techniques, such as AC-matching or term indexing modulo AC[1]. Also, this system is quite easy to adapt to some similar tasks, like matching a pattern $p = \Lambda_i X_i : \tau_i.c$ against a subset of a clause c': we can match $\Lambda_i X_i : \tau_i.(c \vee y)$ against $c' \vee \top$, where $y : o$ is a fresh variable to be matched against the rest of c'. Matching a pattern against a subset of a clause could be useful if the subset is an instance of the negation of the conclusion of a lemma, for instance, because instantiating the lemma would then simplify the clause. The lambda-abstraction is used to have a canonical representation of a pattern (using De Bruijn indexes would also work), so that it can be considered as a constant by Datalog (see section 4). More powerful matching algorithms (e.g., restricted forms of higher-order matching) can be used to find more elaborate instances.

The following property always holds for patterns: if $p = \Lambda_{i=1}^n X_i : \tau_i\ c$, and $a_1 : \tau_1, \dots, a_n : \tau_n$ are terms (in particular, constants), then $p\ a_1\ a_2 \dots\ a_n$ is a well-typed term, that is isomorphic to a concrete first-order clause.

Pattern abstraction allows us to abstract a clause from its concrete local signature. *Pattern instantiation* applies a pattern to a tuple of symbols, returning a concrete clause. Figure 1 describes the following operations (the variable F used for abstraction is assumed to be a fresh variable uniquely associated with the symbol f):

encoding a term or clause into a curried term, noted $\text{enc}(t)$;
decoding a curried term into a term or clause, noted $\text{dec}(t)$;
abstracting a symbol f out of a curried term t, by a variable F, noted $\text{abs}(t, f, F)$;
applying a curried term t to a term a, noted $\text{app}(t, a)$, extended into the n-ary application $\text{app}(t, a_1, \dots, a_n)$.

Encoding is a two steps operation: currying, then prefixing variables with $\mathfrak{v}$ and symbols with $\mathfrak{s}$ to force the matching algorithm to bind abstracted symbols (resp. first-order variables) of the pattern only with symbols (resp. variables) of the clause. Decoding is the exact inverse of encoding.

[1] Our experimental implementation does not implement AC indexing, though.

$$\begin{aligned}
\mathrm{enc}(X) &= \mathfrak{v}\ X \\
\mathrm{enc}(f(t_1,\ldots,t_n)) &= \mathfrak{s}\ f\ \mathrm{enc}(t_1)\ \mathrm{enc}(t_2)\ \ldots\ \mathrm{enc}(t_n) \\
\mathrm{enc}(t_1 \doteq t_2) &= \mathrm{enc}(t_1) \doteq \mathrm{enc}(t_2) \\
\mathrm{enc}(l_1 \vee \ldots \vee l_n) &= \mathrm{enc}(l_1) \vee \ldots \vee \mathrm{enc}(l_n) \\
\mathrm{dec}(\mathfrak{v}\ X) &= X \\
\mathrm{dec}(\mathfrak{s}\ f\ t_1\ \ldots\ t_n) &= f(\mathrm{dec}(t_1),\ldots,\mathrm{dec}(t_n)) \\
\mathrm{dec}(t_1 \doteq t_2) &= \mathrm{dec}(t_1) \doteq \mathrm{dec}(t_2) \\
\mathrm{dec}(l_1 \vee \ldots \vee l_n) &= \mathrm{dec}(l_1) \vee \ldots \vee \mathrm{dec}(l_n) \\
\mathrm{abs}(\mathfrak{s}\ f, f, F) &= \mathfrak{s}\ F \\
\mathrm{abs}(\mathfrak{v}\ X, f, F) &= \mathfrak{v}\ X \\
\mathrm{abs}(t_1\ t_2, f, F) &= \mathrm{abs}(t_1, f, F)\ \mathrm{abs}(t_2, f, F) \\
\mathrm{abs}(t_1 \doteq t_2, f, F) &= \mathrm{abs}(t_1, f, F) \doteq \mathrm{abs}(t_2, f, F) \\
\mathrm{abs}(l_1 \vee \ldots \vee l_n, f, F) &= \lambda F.(\mathrm{abs}(l_1, f, F) \vee \ldots \vee \mathrm{abs}(l_n, f, F)) \\
\mathrm{app}(p, a_1, \ldots, a_n) &= (p\ a_1\ a_2\ \ldots\ a_n) \downarrow^{\beta}
\end{aligned}$$

Fig. 1. Rules for patterns

Example: let us consider the theory of commutative monoids ACU with operator f and neutral element e. Its axioms are (last one is *unit*, or U) :

$$\begin{aligned}
f(X,Y) &= f(Y,X) \\
f(X,f(Y,Z)) &= f(f(X,Y),Z) \\
f(X,e) &= X
\end{aligned}$$

The corresponding patterns, after currying and abstraction, are:

- $\lambda F.(\mathfrak{s}\ F\ (\mathfrak{v}\ X)\ (\mathfrak{v}\ Y) = \mathfrak{s}\ F\ (\mathfrak{v}\ Y)\ (\mathfrak{v}\ X))$
- $\lambda F.(\mathfrak{s}\ F\ (\mathfrak{s}\ F\ (\mathfrak{v}\ X)\ (\mathfrak{v}\ Y))\ (\mathfrak{v}\ Z) = \mathfrak{s}\ F\ (\mathfrak{v}\ X)\ (\mathfrak{s}\ F\ (\mathfrak{v}\ Y)\ (\mathfrak{v}\ Z)))$
- $\lambda F.\lambda E.(\mathfrak{s}\ F\ (\mathfrak{v}\ X)\ (\mathfrak{s}\ E) = \mathfrak{v}\ X))$

Once we have a set of patterns $\mathcal{P} = \{p_1, \ldots, p_n\}$, we can *match* those patterns against clauses of the problem we are trying to solve. Matching a pattern p against a clause c amounts to:

1. choose fresh variables $X_1 : s_1, \ldots, X_n : \tau_n$, where $(\tau_i)_i$ are the input types of the pattern p;
2. compute $t \equiv \mathrm{app}(p, X_1, \ldots, X_n)$;
3. use a matching algorithm modulo AC ($=$ is commutative, and $\vee$ is AC) to match t against $\mathrm{enc}(c)$;
4. for each such matcher σ, its restriction $\sigma' \equiv \sigma|_{\{X_1,\ldots,X_n\}}$ is an instance of p equivalent to c (i.e., $\mathrm{dec}(\mathrm{app}(p, X_1\sigma', \ldots, X_n\sigma')) =_{AC} c$).

Example: let us match the pattern for an identity value, $\lambda F.\lambda E.(\mathfrak{s}\ F\ (\mathfrak{s}\ E) = \mathfrak{s}\ E)$ with $\mathrm{zero} = \mathrm{minus}(\mathrm{zero}, \mathrm{zero})$. We first apply the pattern to fresh variables

F' and E', obtaining after beta-reduction the HO term $\mathfrak{s}\ F'\ (\mathfrak{s}\ E') = \mathfrak{s}\ E'$. The clause is then encoded into $\mathfrak{s}$ zero = $\mathfrak{s}$ minus ($\mathfrak{s}$ zero) ($\mathfrak{s}$ zero); a solution, obtained by AC-matching the equations, is $\sigma = \{F' \mapsto \text{minus zero}, E' \mapsto \text{zero}\}$. This (first-order) instance can only be found thanks to currying.

The next step is to aggregate several pattern instances into an instance of a *theory*, that is, a set of clauses.

4 Meta-Reasoning with Datalog

4.1 Description of an Axiomatic Theory

An *equational theory* is a set of related axioms. Therefore, a *theory pattern* is a set of related clause patterns. We adapt and generalize the mechanism used by Waldmeister[6] for choosing term orderings. In Figure 2, we show a fragment of the file that defines some basic axioms and theories for our prover[2].

```
associative(f) is axiom f(X,f(Y,Z)) = f(f(X,Y), Z).
commutative(f) is axiom f(X,Y) = f(Y,X).
theory ac(f) is associative(f) and commutative(f).
theory aci(f,e) is ac(f) and axiom f(X,e) = X.
```

Fig. 2. Description of Theories

This simple file shows us what is needed to define a theory like AC or the theory of commutative monoids. We need to define some axioms, possibly named, to abstract their symbol out, and to constraint symbols of the axioms to be the same. Indeed, the two clauses $f(f(X,Y),Z) = f(X,f(Y,Z))$ and $g(X,Y) = g(Y,X)$ can be matched, respectively, against the axioms `associative(`f`)` and `commutative(`g`)`, but that does not mean that the theory of AC symbols is present. Those axioms are *parametrized* by symbols f and g.

To be able to constraint symbols in the axioms to be the same, we use the *Datalog* fragment of first-order logic[1]. Datalog only allows function-free Horn clauses, but shows very good computational properties, and a set of Datalog clauses always has exactly one minimal model. We are going to have a Datalog reasoner work on properties of the problem, and communicate with the regular superposition prover.

A Datalog atom is of the form $p(t_1, \ldots, t_n)$ where p is a *predicate symbol* and for all i, t_i is either a *Datalog constant* or a *Datalog variable*. A Datalog clause is a Horn clause, noted a :- $b_1, \ldots b_n$. where a is the conclusion, and b_i are the premises We will write a. for unit clauses. We point out that Datalog constants and variables are nothing like the first-order problem's constants and variables. The trick is that the Datalog reasoner will not "see" what is inside the patterns

[2] The axioms, theories, lemmas and redundancy criteria are defined in a file loaded when the theorem prover starts. The format of the file is defined by a simple grammar that is easy to edit and read, as demonstrated in Figure 2.

it manipulates (such objects are not expressible in Datalog), but it will consider them as blackboxes (constants).

Let us define the black-boxing of patterns, that embeds first-order objects into Datalog constants. Given a pattern p, we write $\lceil p \rceil$ for the boxed version of p; given such a black box b, we define $\lfloor b \rfloor$ its content, obtained by unboxing. Obviously, $\lfloor \lceil p \rceil \rfloor = p$ must hold. Equality over boxes is defined by $\lceil p \rceil = \lceil q \rceil \Leftrightarrow p = q$. The boxing and unboxing functions are trivially extended to any term.

We can now encode properties about the current problem into Datalog atoms, and their definitions into Datalog clauses. Detecting theories requires a few basic properties, which are the following ones:

- Presence of an instance of a pattern, with the corresponding symbols, such as $\mathrm{app}(p, \mathrm{plus})$ where $p \equiv \lambda F.(\mathfrak{s}\ F\ (\mathfrak{v}\ X)\ (\mathfrak{v}\ Y) = \mathfrak{s}\ F\ (\mathfrak{v}\ Y)\ (\mathfrak{v}\ X))$. It is needed for recognizing the constituting axioms of a theory;
- Presence of a named pattern, for instance `commutative(plus)` (which corresponds to the previous pattern). This is used mainly for the user to attach a meaningful name ("associative") to a pattern;
- Presence of an instance of a theory, with the corresponding symbols, for instance `monoid(plus, zero)`;
- Other properties can be encoded (see Sections 4.4 and 5) for more advanced uses of the Datalog reasoner. This makes the detection mechanism quite generic and modular since it allows to define additional properties based on the previously defined ones.

4.2 Encoding of Properties

Such properties are encoded using a distinct Datalog predicate symbol for each kind of property. This way, new properties can be encoded just by reserving a new predicate symbol for them. The basic properties are encoded by:

pattern: A pattern instance $\mathrm{app}(p, a_1, \dots, a_n)$, is encoded using the predicate "pattern", into $\mathtt{pattern}(\lceil p \rceil, \lceil a_1 \rceil, \dots, \lceil a_n \rceil)$

theory: A theory is a name, parametrized by a set of function symbols; A theory instance "name"$(a_1, \dots, a_n)$ is encoded using the predicate "theory" into $\mathtt{theory}(\lceil \text{"name"} \rceil, \lceil a_1 \rceil, \dots, \lceil a_n \rceil)$;

named pattern: It is similar to a theory with a single axiom, but using the predicate "axiom"; so, for instance, $\mathtt{associative}(f)$ is encoded into $\mathtt{axiom}(\lceil \text{"associative"} \rceil, \lceil f \rceil)$.

4.3 Encoding of Definitions

It is also necessary to define theories and named patterns, by Datalog clauses that will trigger a property when the constitutive patterns of the theory (respectively named patterns) are present. This requires Datalog variables; if a theory (named $\mathcal{N}$) is defined by $\mathcal{N}(f_1, \dots, f_n) \equiv p_1, \dots, p_m$ (with m premises), its definition will be a Datalog clause with m Datalog atoms as premises. Let us map $f_1, \dots, f_n$ to fresh Datalog variables $F_1, \dots, F_n$. The premises p_i are translated to Datalog atoms q_i as follows:

- If p_i expresses the presence of a named pattern or a theory $\mathcal{N}'(f_{\sigma(1)}, \ldots, f_{\sigma(k)})$ parametrized by k symbols, then $q_i = \texttt{axiom}(\lceil \mathcal{N}' \rceil, F_{\sigma(1)}, \ldots, F_{\sigma(k)})$ or $q_i = \texttt{theory}(\lceil \mathcal{N}' \rceil, F_{\sigma(1)}, \ldots, F_{\sigma(k)})$
- If p_i expresses the presence of a pattern instance app$(p, f_{\sigma(1)}, \ldots, f_{\sigma(k)})$, with k symbols as parameters, then $q_i = \texttt{pattern}(\lceil p \rceil, F_{\sigma(1)}, \ldots, F_{\sigma(k)})$

The definition is simply $\texttt{theory}(\lceil \mathcal{N} \rceil, F_1, \ldots, F_n)$:- $q_1, \ldots, q_m$. (easily adapted for named patterns).

4.4 Encoding of Other Properties

Other properties, depending on how the prover uses knowledge about theories, can be encoded the same way. For instance, if we want to gather information specifically about AC symbols, we can use a fresh Datalog predicate "ac" and the following clause:

$$\texttt{ac}(F) \text{ :- } \texttt{theory}(\lceil \text{“ac”} \rceil, F)$$

Then, whenever a Datalog fact $ac(a)$ is found by the Datalog reasoner, we know that $\lfloor a \rfloor$ is an associative commutative symbols in the current problem (and we can activate a special strategy to deal with it in the refutational theorem prover, like superposition modulo AC). We will see more detailed examples in Section 5.

4.5 Incremental Computation

When an automated theorem prover tries to solve a given problem, some properties of this problem may not be readily available for recognition. Instead, it may take some time to reach some axioms that are part of a theory's definition. Therefore, *incrementality*, i.e. the ability to discover properties and deduce other properties during the process of solving the problem, is crucial.

Our implementation of the Datalog reasoner therefore does not provide a query interface, but rather an incremental interface; clauses can be added one by one, each time updating the current set of clauses (saturated under the *immediate consequence operator*). The immediate consequence operator adds $a\sigma$ to the set of facts, if a :- $b_1, \ldots, b_n$. is a clause and for all $i \in \{1 \ldots n\}$, $b_i\sigma$ belongs to the set of facts. Because we only work with *safe clauses*, i.e., clauses in which $\text{vars}(a) \subseteq \bigcup_{i=1}^{n} \text{vars}(b_i)$, we are sure that $a\sigma$ is ground.

To achieve incremental computation, the Datalog reasoner is based on unit resolution with selection. Every non-unit clause, of the form a :- $\underline{b_1}, b_2, \ldots, b_n$. with $n > 0$, gets its first body literal *selected* (the underlined literal). Only one inference rule is needed to ensure completeness:

$$\frac{a \text{ :- } \underline{b_1}, b_2, \ldots, b_n. \qquad c. \qquad b_1\sigma = c\sigma}{a\sigma \text{ :- } \underline{b_2\sigma}, b_3\sigma, \ldots, b_n\sigma.}$$

Every time we add a clause to the Datalog reasoner, the resolution rule is applied between clauses of the current fixpoint, and the new clause. Callbacks can be attached to the Datalog reasoner, to be called whenever a new fact is deduced by this inference rule; the new facts are then added to the reasoner one by one – with their own chance to trigger inferences. A non-perfect discrimination tree is used to index selected literals and facts, in order to make this inference reasonably efficient.

4.6 Backward Chaining

In some cases, the underlying first-order calculus used by the theorem prover may never discover some axioms (like associativity). This is the case, for instance, for refutational provers based on resolution or superposition, because they may not need to infer the axiom to remain complete, in case it is deducible from the initial problem but redundant. In this case, if, for instance, out of the 10 axioms that are necessary for an instance of a theory to hold, 9 are present, the Datalog prover may pro-actively spawn a sub-prover to try to show this axiom.

The Datalog reasoner can use prolog-like backward chaining to find which literals may help finding new facts. Assuming we keep a set $\mathcal{G}$ of *goals* – a goal being a literal whose instances may help solving already existing goals – the following rule updates the set of goals:

$$\frac{a \text{ :- } \underline{b_1}, b_2, \ldots, b_n. \qquad g \in \mathcal{G} \qquad a\sigma = g\sigma}{b_1\sigma \in \mathcal{G}}$$

We did not implement a system that spawns sub-provers that attempt to show missing axioms, but it would be quite simple by finding which of the current *goals* of the Datalog reasoner belong to the category of totally instantiated patterns (where all parameters of the pattern are constants, not Datalog variables). However, goals are already important in our implementation, because we only try to match against concrete clauses, the patterns which are currently goals in the Datalog reasoner. In other words, clause patterns we want to match with concrete clauses are $\{p \mid \texttt{pattern}(\lceil p \rceil, F_1, \ldots, F_n) \in \mathcal{G}\}$. The initial set of goals is the set of conclusions of clauses, that is, $\mathcal{G}_0 = \{a \mid a \text{ :- } \underline{b_1}, b_2, \ldots, b_n.\}$, but we could choose a different (restricted) set of goals; for instance, if some lemmas hold only when arithmetic symbols are present, their conclusion shall not be goals until an arithmetic formula is detected, not to clutter the pattern recognition mechanisms.

5 Why Recognize Theories?

In previous sections, we explained how to recognize individual axioms and theories during a saturation proof search. We will now give some ways to use this knowledge about the problem at hand. Of course, because the coupling with the Datalog reasoner is modular, one can make any use she wants from the output of the Datalog reasoner, and even add whichever clauses and predicates she judges useful. For most uses of the Datalog reasoner, we use a dedicated predicate, and some clauses whose premises are theories or individual axioms. We will call *Knowledge Base* the set of definitions and facts that is given to the theorem prover when it starts; it should contain definitions of axioms, theories, and other data that is specific to how we use knowledge about the problem.

5.1 Lemmas

Let us call *lemma* an already proven logic statement of the form "clause a is true if clauses $b_1, \dots, b_n$ are". Such a lemma may be a mathematical result that the user makes available to the theorem prover, or some previously proven theorem; all we need to know is that this statement is already known to be proven. It can be encoded in Datalog by abstracting symbols from the conclusion and the premises (see section 4.3).

Our current *Knowledge Base* contains only one lemma that we added by hand, but it has shown to be quite useful. We call this lemma *un-mangling of functional relations*. Given the two properties about a ternary relation symbol r and a binary function symbol f:

- $\texttt{functional}(r) \equiv r(X, Y, Z) \wedge r(X, Y, Z') \Rightarrow Z = Z'$;
- $\texttt{total}(r, f) \equiv r(X, Y, f(X, Y))$.

We know that r encodes the graph of the function f. Hence the following lemma, that states $r(X, Y, Z) \Leftrightarrow (f(X, Y) = Z)$. Its definition is shown in Figure 3. Such a lemma, if applied during the preprocessing phase, allows one to unfold the definition of r, removing it from the problem (our prover uses the calculus of [5], which allows one to use such equivalences for rewriting). After unfolding of r, the problem is "more equational": axioms such as the commutativity of f, that were encoded by $r(X, Y, Z) \wedge r(Y, X, Z') \Rightarrow Z = Z'$, become $f(X, Y) = f(Y, X)$ after simplifications. An equational theorem prover such as E[11] will be able to use more rewriting-based simplification rules.

```
functional(r) is axiom ~r(X,Y,Z) | ~r(X,Y,Z2) | Z=Z2.
total(r, f) is axiom r(X,Y,f(X,Y)).
lemma r(X,Y,Z) <=> Z = f(X,Y) if functional(r) and total(r, f).
```

Fig. 3. Un-mangling of Functional Relations Lemma

Before we developed the general theory detection system, specific code was dedicated to recognizing instances of this lemma in Zipperposition. The code took around 85 lines of OCaml and would only work on initial axioms. We emphasize the fact that detecting instances of this lemma require many features, like detecting non-unit clauses with several abstracted symbols (f and r), and then joining the multi-symbol axioms together. Now, to add similar lemmas, we only need a few lines in the previously mentioned declarative syntax. Each lemma is encoded as exactly one Datalog clause, whose conclusion is a pattern instance.

5.2 Equational Redundancy Criteria

As the authors of [2] point out, superposition-based theorem provers such as SPASS[14], E[11] or Vampire[9] can quickly become overwhelmed by the amount of clauses that are generated in the presence of equational theories such as AC, ACI or other algebraic structures. This is exacerbated by the fact that superposing with commutativity is very often possible in both ways, since the axiom is not oriented by the usual KBO and RPO term orderings. A lot of efforts [12] [3] have been devoted in extending the superposition calculus to work modulo AC, or modulo theories that encompass AC; however it is usually delicate both to implement and prove complete each instance of superposition modulo a theory. We expose here a way of using some knowledge about equational theories to prune the search space of such theorem provers. We will use some definitions and theorems from [2].

Redundancy Criterion. Let use consider the section 5 of [2]. A ground convergent system of equations $R_0 \cup E_0$ is used to decide of the AC theory for some symbol f. The Theorem 5.1 states that any equation $s = t$, not part of the system $R_0 \cup E_0$, where $s =_{AC(f)} t$, is redundant and can be disposed of during the proof search process. Let us examine how this theorem is proved:

$R_0(E_0)$ (the set of orientable instances of the equations) is terminating by construction. For every critical pair, all of its ground instances are joinable. Hence $R_0(E_0)$ is confluent on Term(F^e). Consequently, if $s =_{R_0 \cup E_0}$, then $s\sigma \downarrow t\sigma$ for any ground substitution σ, and therefore $s \Downarrow^{\triangleright} t$.

We can adapt this proof, for a given reduction ordering $\succ$, to any set of equations E that is ground convergent. Indeed, with the trivial rewriting system $R = \emptyset$, $R(E)$ is ground convergent and terminating (included in the well-founded reduction ordering $\succ$). Let us consider an equation $s = t$, and write $s' = t'$ the same equation where variables $x_0, \ldots, x_n$ are replaced by fresh constants $c_0, \ldots, c_n$; let us extend the ordering $\succ$ to a reduction ordering $\succ'$ that contains $\succ$ ([2] explains how to do it for LPO and KBO, respectively, in lemmas 5.2 and 5.3). Then, if $s' \downarrow^{R(E)^{\succ'}} t'$, every ground instance of $s = t$ is joinable by $E^{\succ}$, and $s = t$ is redundant.

In other words, E provides us with a redundancy criterion for any equation $s = t$, by checking whether s' and t' have the same normal form. In practice, we just have to consider variables in s and t as constants, extend $\succ$ with those new constants, and compute the normal form of both terms w.r.t. orientable equations of E. The resulting simplification rules are exposed in Figure 4. The double bar indicates that the clause above is *replaced* by the clause below. The operation const replaces variables in s and t by fresh constants; ground joinability of s and t is then implied by joinability of const(s) and const(t).

Theory Combination. If several theories $T_1, \ldots, T_n$ occur in a single problem, then we can combine several ground confluent systems $E_1, \ldots, E_n$. The combination will always be terminating (because included in $\succ$), but not necessarily ground-convergent any more. However, if the theories have disjoint signatures,

Tautology Deletion modulo $\mathcal{T}$:

$$\frac{s = t \lor C}{} \ \text{if } \mathrm{const}(s) \downarrow^{\mathcal{T}} \mathrm{const}(t)$$

Equality Resolution modulo $\mathcal{T}$:

$$\frac{s \neq t \lor C}{C} \ \text{if } \mathrm{const}(s) \downarrow^{\mathcal{T}} \mathrm{const}(t)$$

Fig. 4. Simplification Rules for the Redundancy Criterion on a Theory $\mathcal{T}$

the combination is still a decision procedure on terms that are exclusively composed of free symbols and symbols from T_i for some i. On mixed term, we may want to purify terms by introducing fresh constants for subterms that belong to a different theory.

Interaction with Datalog. Now that we have a redundancy criterion for some theories, we can encode it, regardless of the concrete signature, into Datalog clauses. The encoding is more complicated than previous ones — it involves boxing several patterns, keeping track of a relationship between Datalog variables and the symbols of each equation of a ground joinable system — but it follows the same principles. Then, such a redundancy criterion can be triggered when the theory it decides is detected; the equational theorem prover can then use it, if its ordering is compatible. The E[11] prover does exactly this, for the specific case of AC symbols (with the same rules as in Figure 4 where $\mathcal{T}$ is replaced by AC for a set of symbols).

Computing Criteria for a Theory. If we want to compute such a ground convergent system of equations E for a given theory E_0 (for instance, AC or a formulation of Group theory), a possibility is to use the *syntactic criterion* described in Theorem 5.2 of [2] in order to saturate E_0 (in a given ordering $\succ$), while discarding ground joinable equations. If this process terminates, it yields a set E that must be ground convergent (otherwise there would be a non ground-joinable equation in E).

Given an equational theory E_0 that has symbols $f_1, \ldots, f_n$, we can try to compute such ground convergent systems by saturation for a finite set of LPO and KBO orderings on those symbols. Whenever a saturation succeeds for some ordering $\succ$, it yields a decision procedure E for E_0 in $\succ$. If we later meet a problem where the axioms contain E_0, and the ordering is compatible with $\succ$, we add E to the set of clauses and remove any clause $c \notin E$ that is ground-joinable by $E^{\succ}$.

5.3 Term Orderings

In [6], the authors describe a system, quite similar to ours, used by Waldmeister during the preprocessing phase to detect some theories and heuristically choose a term ordering that experience has shown to be efficient for those theories. This kind of analysis of the problem is feasible with our approach as well. However, since theories can be detected during the proof search, information on the ordering may come too late; in this case, *restarting* the prover with a different ordering, chosen with more information about the problem — maybe keeping some useful deduced clauses, like rewriting rules — can be relevant.

6 Experimental Results

We compared our experimental implementation[3] (version 0.2) with SPASS[14] and E[11] on categories RNG and GRP of the TPTP[13] base of problems. Benchmarks include both `zipperposition` — our theorem prover with theory detection, relational un-mangling lemma, and redundancy criteria (for AC, commutative monoids and abelian groups) — `zipperposition-lemma`, with the relational lemma but no redundancy criteria, and `zipperposition-no-theories`, in which all theory handling is disabled. The results are exposed in Figure 5. Overall, on 1434 problems, `zipperposition` proves 7 problems that are not proven by SPASS nor E within the 120s timeout. Zipperposition is able to detect at least one theory in 594 problems out of 1434, and triggers the lemma in 68 problems. Among the 594 problems with theories, 31 are solved by *zipperposition* or *zipperposition-lemma*, but not by *zipperposition-no-theories*, and 7 are solved by the latter but not by the former (because the prover was slower or it pruned the wrong part of the search space). This ratio becomes 7 to 2 on the problems in which the lemma is applied.

We can already see that the redundancy criterion, with its quite naive implementation, already brings benefits. The un-mangling lemma makes a significant difference on the set of problems in which it applies. On individual problems, the difference can be striking: some problems that would not terminate within 2 minutes become trivial enough to get solved in 0.5s when lemma detection is enabled. Those results are encouraging, and we believe that using a meta-prover may find more uses in automated theorem proving. Profiling shows that the Datalog reasoner represents a negligible fraction of the run-time (less than 1%). On the other hand, our implementation is more naive and less efficient than SPASS or E (which have a more powerful calculus, better heuristics, or a more efficient implementation), which can explain why they still solve more problems. Our technique could be integrated in other theorem provers to discover lemmas or usable redundancy criteria — especially for scheduling provers (like iProver[7]) because meta-level facts that are discovered during a time slice can be used for the next ones (using a suitable term ordering, etc.).

Three problems are solved only by the versions of Zipperposition that use lemma detection: GRP392-1.p, GRP393-1.p and GRP394-1.p. Interestingly, all three are satisfiable problems in relational form where the un-mangling lemma

[3] We point out that our implementation of superposition is not nearly as good as SPASS or E, which are the result of years of work.

Prover	Proved	success rate(%)	proved /594	%	proved /68	%
E	1047	73.0	430	72.4	59	86
SPASS	863	60.1	376	63.3	50	73
zipperposition	531	37.0	202	34.0	56	82
zipperposition-lemma	527	36.7	199	33.5	57	83
zipperposition-no-theories	504	35.1	191	32.1	52	76

Fig. 5. Benchmark Results: Number of Solved Problems

transforms into easily saturated sets of equations. This is only possible because the calculus of [5] turns some equivalences into rewrite rules.

Conclusion and Possible Extensions

We have shown a generic and flexible way to detect instances of axioms and theories during the search for a (clausal) proof. The use of a Datalog incremental inference system, which manipulates assertions about the problem itself, makes the meta-level reasoning flexible, modular and allows to have one's own meta-facts (properties) triggered by new meta-assertions. This technique already shows very promising results, and can be improved further with more sophisticated uses of the detected theories. We believe that this kind of combination, although still quite simple, bears some resemblance with the way real mathematicians solve problems. Using several levels of description and proof may also help making automated proofs more understandable, saturation proofs being often blamed for being very unintuitive to human users. Further development includes:

- making the reasoner more proactive by having it spawning subprocesses to try to prove missing axioms;
- computing redundancy criteria for equational theories off-line. Theories could be extracted from axiom files, before a redundancy criterion is looked for by saturating the axioms;
- automatically extract lemma from successful proofs in order to help solving similar problems;
- implementing this technique in a state of the art prover.

Acknowledgements. We would like to thank Gilles Dowek for his help, and the anonymous reviewers for their detailed and helpful comments.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Avenhaus, J., Hillenbrand, T., Löchner, B.: On using ground joinable equations in equational theorem proving. Journal of Symbolic Computation 36(1-2), 217–233 (2003)

3. Bachmair, L., Ganzinger, H.: Associative-commutative superposition. In: Dershowitz, N., Lindenstrauss, N. (eds.) CTRS 1994. LNCS, vol. 968, pp. 1–14. Springer, Heidelberg (1995)
4. Denzinger, J., Schulz, S.: Learning domain knowledge to improve theorem proving. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 62–76. Springer, Heidelberg (1996)
5. Ganzinger, H., Stuber, J.: Superposition with equivalence reasoning and delayed clause normal form transformation. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 335–349. Springer, Heidelberg (2003)
6. Hillenbrand, T., Jaeger, A., Löchner, B.: System description: Waldmeister – improvements in performance and ease of use. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 232–236. Springer, Heidelberg (1999)
7. 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)
8. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning. Elsevier, MIT Press (1999)
9. Riazanov, A., Voronkov, A.: Vampire 1.1 (System description). In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 376–380. Springer, Heidelberg (2001)
10. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. J. ACM 12(1), 23–41 (1965)
11. Schulz, S.: E - a brainiac theorem prover. AI Commun. 15(2,3), 111–126 (2002)
12. Stuber, J.: Superposition theorem proving for abelian groups represented as integer modules. In: Ganzinger, H. (ed.) RTA 1996. LNCS, vol. 1103, pp. 33–47. Springer, Heidelberg (1996)
13. 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)
14. Weidenbach, C., Schmidt, R.A., Hillenbrand, T., Rusev, R., Topic, D.: System Description: SPASS Version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 514–520. Springer, Heidelberg (2007)

Mechanizing the Metatheory of Sledgehammer

Jasmin Christian Blanchette and Andrei Popescu

Fakultät für Informatik, Technische Universität München, Germany

Abstract. This paper presents an Isabelle/HOL formalization of recent research in automated reasoning: efficient encodings of sorts in unsorted first-order logic, as implemented in Isabelle's Sledgehammer proof tool. The formalization provides the general-purpose machinery to reason about formulas and models, emulating the theory of institutions. Quantifiers are represented using a nominal-like approach designed for interpreting syntax in semantic domains.

1 Introduction

Despite steady progress in the usability of proof assistants, paper proofs reign supreme in the automated reasoning community. Myreen and Davis's verification of an ACL2-like prover in HOL4 [17] and Harrison's partial self-verification of HOL Light [13] are exceptions rather than the rule. Important metamathematical results have been formalized (e.g., Shankar's Gödel proof [26]), but new research is still carried out almost exclusively on paper, with all the risks this entails.

This paper presents a formalization in Isabelle/HOL [18] of the proofs for translations from many-sorted to unsorted first-order logic (FOL). Claessen et al. [10] designed lightweight encodings that eliminate much of the clutter associated with traditional schemes. Blanchette et al. [3, 4] introduced even lighter encodings in a sequel. Central to these new encodings is the notion of monotonicity. Informally, a sort is monotonic if its domain can be extended with new elements without compromising satisfiability. Nonmonotonic sorts can be made monotonic by introducing protector functions or predicates, and monotonic sorts can be merged into a single sort.

Sorts are trivially monotonic in FOL without equality. The addition of interpreted equality makes it possible to encode upper cardinality bounds on the models, breaking monotonicity. Like other interesting semantic properties, monotonicity is undecidable but can often be inferred in practice. Monotonicity has many applications in theorem provers and model finders [5, 10]. It is also roughly equivalent to smoothness, a criterion that arises when combining decision procedures in SMT solvers [28].

The Sledgehammer [19] proof tool for Isabelle/HOL relies on the monotonicity-based encodings to apply state-of-the-art unsorted provers to sorted problems. The tool translates interactive proof goals along with relevant lemmas and invokes the external automatic theorem provers to find proofs, which are reconstructed through Isabelle's inference kernel. Early versions of Sledgehammer relied on unsound sort encodings; as a result, they would often find spurious, unreconstructable proofs, which annoyed users and could conceal sound proofs. Whereas Sledgehammer reconstructs the external proofs, tools such as Monotonox [10] and the fully-automatic competition version of Isabelle [27] do not perform such checks; soundness is crucial for them.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 245–260, 2013.

The mechanization of the sort encodings fully covers the correctness proofs from Claessen et al. [10] and the monomorphic half of its sequel [3, 4], as well as a theorem by Bouillaguet et al. [9]. This formalization work arose from a desire to provide more solid assurance to this recent research. Even if the intuition is clear, a paper proof offers many opportunities for flaws, especially because of the variety of encodings.

The mechanization effort partly coincided with the development of the informal proofs [4]. The two proofs largely follow the same conventions, with one major difference: The core of the formal proof (Sections 3 to 5) assumes quantifier-free clausal normal form (CNF) rather than negation normal form (NNF). This reduces the exposure to name binders, which are notoriously difficult to reason about. The results are lifted to NNF using a clausification theorem (Sections 6 to 8). This organization is reminiscent of the architecture of automatic reasoners that combine a clausifier and a CNF core.

Isabelle's higher-order logic (HOL) might not be as expressive as set or type theory, but it can cope with the statements and proofs of classical metatheorems (as shown by Harrison and others [2, 12, 25]) and practical results. The proof assistant offers many conveniences; two features have been particularly useful:

- *Locales* [1,15] parameterize theories over constants and assumptions, with the usual benefits associated with modularity. Locales are particularly suited to expressing logic translations abstractly as in the theory of institutions [11].
- A framework for *syntax with bindings* [23,24] eases reasoning about quantified formulas. It lies at the intersection of first-order nominal approaches [21] and higher-order abstract syntax [20]. The framework is designed specifically for interpreting syntax in semantic domains.

Locales have been part of Isabelle for many years and are widely used. The syntax with bindings is a newer addition; the current application is among the first case studies that feature it. The formal proofs are available online [6, 7].

Although sort encodings are the focus of this paper, our infrastructure is designed to be reusable for other applications of many-sorted FOL. Many important metatheories are awaiting formalization, such as the completeness of paramodulation and tableaux.

2 An Isabelle View of Logic Translations

The formalization covers a variety of translations, including not only the sort encodings but also clausification. The guiding principles, described below, originate from the theory of institutions; their Isabelle materialization relies on locales.

Institutions. A *logic* $\mathcal{L}$ provides a category of signatures $\mathcal{Sig}$ and, for each signature $\Sigma \in \mathcal{Sig}$, a set of sentences $\mathcal{Sen}(\Sigma)$, a class of structures (interpretations) $\mathcal{Str}(\Sigma)$, and a satisfaction relation $\models_\Sigma$ between structures and sentences. A signature morphism $k : \Sigma \to \Sigma'$ is equipped with a forward sentence translation $k : \mathcal{Sen}(\Sigma) \to \mathcal{Sen}(\Sigma')$ and a backward structure translation $\upharpoonright_k \,:\, \mathcal{Str}(\Sigma') \to \mathcal{Str}(\Sigma)$. An *institution* is a logic whose signature morphisms enjoy the property that "truth is invariant under change of notation": $\mathcal{M}' \models_{\Sigma'} k\,\varphi \longleftrightarrow \mathcal{M}' \!\upharpoonright_k\, \models_\Sigma \varphi$ for all $k : \Sigma \to \Sigma'$, $\mathcal{M}' \in \mathcal{Str}(\Sigma')$, and $\varphi \in \mathcal{Sen}(\Sigma)$.

A translation of $\mathcal{L}$-problems (sets of sentences) into $\mathcal{L}'$-problems consists of a function $\$$ between $\mathcal{L}$'s and $\mathcal{L}'$'s signature classes and, for each $\Sigma \in \mathrm{dom}(\$)$ and Σ-problem Φ, a sentence translation $\mathsf{enc}_\Phi : \mathcal{S}en(\Sigma) \to \mathcal{S}en(\Sigma^\$)$ and a set of axioms $\mathcal{A}x_\Phi \subseteq \mathcal{S}en(\Sigma^\$)$. The translation of Φ is defined as $\mathsf{enc}\ \Phi = \{\mathsf{enc}_\Phi\ \varphi \mid \varphi \in \Phi\} \cup \mathcal{A}x_\Phi$. Thus, $\mathcal{L}$-problems are mapped to $\mathcal{L}'$-problems by joining an elementwise translation and additional axioms. Given a class $\mathcal{C}$ of $\mathcal{L}$-problems, the translation is *sound* w.r.t. $\mathcal{C}$ if satisfiability of Φ implies satisfiability of $\mathsf{enc}\ \Phi$ for all $\Phi \in \mathcal{C}$, and *complete* if the converse holds.

The institution literature focuses on "uniform" encodings. For these, the sentence translation depends only on Φ's signature Σ, and there exists a backward translation $\mathsf{dec} : \mathcal{S}tr(\Sigma^\$) \to \mathcal{S}tr(\Sigma)$ for which an inter-institution version of the institutional condition holds: $\mathcal{M}' \models_{\Sigma^\$} \mathsf{enc}_\Sigma\ \varphi \longleftrightarrow \mathsf{dec}\ \mathcal{M}' \models_\Sigma \varphi$. This condition implies completeness.

The source logic $\mathcal{L}$ for all the translations considered in this paper is many-sorted FOL; the target logic $\mathcal{L}'$ is either many-sorted or unsorted FOL. Sentences are either CNF clauses or NNF formulas. Most of the translations are nonuniform.

Isabelle. Isabelle/HOL is based on polymorphic HOL, which can be thought of as a fragment of Standard ML enriched with logical constructs and a proof system. Type variables are identified by a leading prime (e.g., $'a$). The type $\sigma \to \tau$ is interpreted as the set of (total) functions from σ to τ. Propositions are terms of type bool, and predicates are functions to bool. Function applications are written without parentheses (e.g., $f\ x\ y$) or in infix notation (e.g., $x + y$). Constants and variables can be functions.

The type $'a\ \mathsf{list}$ of finite lists over $'a$ is generated freely from the empty list $[]$ and the infix constructor $\# : {'a} \to {'a}\ \mathsf{list} \to {'a}\ \mathsf{list}$. The notation $[x_1, x_2, \ldots, x_n]$ abbreviates $x_1 \# (x_2 \# (\cdots \# (x_n \# []) \cdots))$. The higher-order constant $\mathsf{map} : ({'a} \to {'b}) \to {'a}\ \mathsf{list} \to {'b}\ \mathsf{list}$ applies a unary function to each element in a list, and $\mathsf{set} : {'a}\ \mathsf{list} \to {'a}\ \mathsf{set}$ returns the set of elements in a list. Sets are written using traditional mathematical notation. Type parameters of polymorphic types are sometimes omitted (e.g., set for $'a\ \mathsf{set}$).

Locales. Isabelle locales are a structuring mechanism provided on top of basic HOL. They fix types, constants, and assumptions, as in the following schematic examples:

$$\texttt{locale}\ \mathsf{X} = \texttt{fixes}\ {'a}\ \texttt{fixes}\ \mathsf{c} : \sigma_{'a}\ \texttt{assumes}\ P_{'a,\mathsf{c}}$$
$$\texttt{locale}\ \mathsf{Y} = \texttt{fixes}\ {'b}\ \texttt{fixes}\ \mathsf{d} : \tau_{'b}\ \texttt{assumes}\ Q_{'b,\mathsf{d}}$$

The definition of locale X fixes a type $'a$ and a constant c whose type $\sigma_{'a}$ may depend on $'a$, and states an assumption $P_{'a,\mathsf{c}} : \mathsf{bool}$ over $'a$ and c. Lemmas proved within the locales can rely on them. In general, a single locale can introduce several types, constants, and assumptions. The definition of X also produces a polymorphic *locale predicate* $\mathsf{X} = (\lambda c.\ P_{'a,c})$. Seen from outside the locale, the lemmas proved in locale X are polymorphic in type variable $'a$, universally quantified over variable c, and conditional on $\mathsf{X}\ c$.

Locales support inheritance, union, and embedding. To embed X into Y, one needs to indicate how an arbitrary instance of X can be regarded as an instance of Y, by providing, in the context of X, definitions of the types and constants of Y together with proofs of Y's assumptions. The command

$$\texttt{sublocale}\ \mathsf{X} < \mathsf{Y}\ \texttt{where}\ {'b} = \upsilon\ \texttt{and}\ \mathsf{d} = t$$

emits the goal $Q_{\upsilon,t}$, where υ and t may depend on types and constants from X. After the proof, all the lemmas proved in the Y become available in X, with υ and t in place of $'b$ and d. Homonymous constants d in X and Y are instantiated as $\mathsf{d} = \mathsf{d}$ by default.

The sublocale relationship is sometimes abbreviated to $\mathsf{X}_{'a,\mathsf{c}} < \mathsf{Y}_{v,t}$ or $\mathsf{X} < \mathsf{Y}$.

Locales provide a shallow realization of institutions in Isabelle. The institutional methodology serves as an inspiration and guidance to formulate results about specific logic translations in a consistent style. Given a logic $\mathcal{L}$, its signatures $\mathcal{S}ig$ are captured by a locale $\mathcal{L}$.Signature, which fixes Isabelle constants for the signature components (e.g., sorts and symbols) and defines a notion of sentence (e.g., clauses or formulas). A locale $\mathcal{L}$.Problem extends $\mathcal{L}$.Signature with a fixed set of sentences Φ. Structures $\mathcal{M}$ are represented by a locale $\mathcal{L}$.Structure that also defines a notion of satisfaction. Finally, satisfiable problems are represented by a locale $\mathcal{L}$.Model that joins $\mathcal{L}$.Problem and $\mathcal{L}$.Structure and further requires satisfaction between Φ and $\mathcal{M}$.

In this setting, translations between logics $\mathcal{L}$ and $\mathcal{L}'$ and their properties are captured via locale embedding mechanisms in four steps.

SIG: Define \$ as a sublocale relationship $\mathcal{L}$.Signature < $\mathcal{L}'$.Signature with suitable parameter instantiations reflecting the definition of $\Sigma^{\$}$ in terms of Σ.

TRANS: Define enc_Φ inside $\mathcal{L}$.Problem (where Σ and the Σ-problem Φ are fixed).

SOUND: To prove soundness, define a $\Sigma^{\$}$-structure $\mathcal{M}'$ inside $\mathcal{L}$.Model (where the signature Σ, the Σ-problem Φ, and the structure $\mathcal{M}$ such that $\Phi \models_\Sigma \mathcal{M}$ are fixed) and show $\mathcal{L}.\mathsf{Model}_{\mathcal{M}} < \mathcal{L}'.\mathsf{Model}_{\mathcal{M}'}$.

COMPLETE: To prove completeness, define a locale Problem_Model′ = $\mathcal{L}$.Problem + $\mathcal{L}'$.Model that joins a Σ-problem Φ and a $\Sigma^{\$}$-model $\mathcal{M}'$ of enc Φ, define inside Problem_Model′ a Σ-structure $\mathcal{M}$, and show $\mathsf{Problem\_Model}'_{\mathcal{M}'} < \mathcal{L}.\mathsf{Model}_{\mathcal{M}}$.

3 Clausal First-Order Logic

The terms, atoms, and literals of (quantifier-free) CNF are represented in HOL by ML-style free datatypes, parameterized by types $'f$ and $'p$ of function and predicate symbols:

```
datatype 'f tm =          datatype ('f,'p) atm =      datatype ('f,'p) lit =
  Var var |                 Pr 'p ('f tm list) |        Pos (('f,'p) atm) |
  Fn 'f ('f tm list)        Eq ('f tm) ('f tm)          Neg (('f,'p) atm)
```

The type var is countably infinite. An atom is either an applied predicate (e.g., $p(t)$) or equality (e.g., $t \approx u$). A clause is a list of literals (interpreted disjunctively), and a problem is a set of clauses (interpreted conjunctively). Formally, $('f,'p)$ clause = $('f,'p)$ lit list and $('f,'p)$ problem = $('f,'p)$ clause set. The CNF representation involves no name binders, unlike (quantified) NNF (Section 6).

Many-sorted signatures (for CNF and NNF) are captured by the following locale:

```
locale Signature =
  fixes 's and 'f and 'p
  fixes arity_F : 'f → 's list and res : 'f → 's and arity_P : 'p → 's list
  assumes countable UNIV_'s and countable UNIV_'f and countable UNIV_'p
```

The locale is parameterized by types for sorts ($'s$), function symbols ($'f$), and predicate symbols ($'p$), all required to be countable (i.e. finite or countably infinite). The locale attaches to each symbol a sort arity ($\mathsf{arity}_\mathsf{F}$ or $\mathsf{arity}_\mathsf{P}$) and, for functions, a result sort (res). The sort arity can be empty. Symbols cannot be overloaded. The polymorphic constant $\mathsf{UNIV}_{'a} : {'a}$ set is predefined in Isabelle as the set of all values of type $'a$.

The Signature locale defines an underspecified function sort : $\mathsf{var} \to {}'s$ that arbitrarily assigns sorts to variables. Whereas the formalization consistently refers to FOL's sorts as types (in view of a possible extension to n-ary type constructors and polymorphism), in this paper they are more precisely called sorts. Wellsortedness and wellformedness of terms and the other syntactic categories are defined in the usual way. Wellformedness is a precondition to many operations, but such details are omitted here.

The Problem locale joins a signature Σ and a CNF Σ-problem Φ. The Structure locale combines a signature, a universe $'u$, and a triple of functions $(\mathsf{int_S}, \mathsf{int_F}, \mathsf{int_P})$ that interpret sorts, function symbols, and predicate symbols:

$$\begin{array}{l} \texttt{locale}\ \mathsf{Problem} = \mathsf{Signature}_{'s,'f,'p}\ \mathsf{arity_F}\ \mathsf{res}\ \mathsf{arity_P} + \\ \quad \texttt{fixes}\ \Phi : ('f,'p)\ \mathsf{problem} \end{array}$$

$$\begin{array}{l} \texttt{locale}\ \mathsf{Structure} = \mathsf{Signature}_{'s,'f,'p}\ \mathsf{arity_F}\ \mathsf{res}\ \mathsf{arity_P} + \\ \quad \texttt{fixes}\ 'u \\ \quad \texttt{fixes}\ \mathsf{int_S} : 's \to 'u \to \mathsf{bool}\ \mathsf{and}\ \mathsf{int_F} : 'f \to 'u\ \mathsf{list} \to 'u\ \mathsf{and} \\ \qquad\qquad \mathsf{int_P} : 'p \to 'u\ \mathsf{list} \to \mathsf{bool} \end{array}$$

A few wellformedness assumptions are made on the triple $(\mathsf{int_S}, \mathsf{int_F}, \mathsf{int_P})$, such as inhabitation of all sorts ($\forall\sigma.\ \exists d.\ \mathsf{int_S}\ \sigma\ d$). The Structure locale also defines the interpretation of terms and satisfaction of clauses. A related locale, Model, represents satisfiable CNF problems by combining a Problem and a Structure it satisfies.

4 Monotonicity and Its Inference

This section focuses on monotonicity in its own right; Section 5 discusses the associated sort encodings. To simplify the monotonicity arguments, both sections assume a fixed infinitely countable type ω as the universe $'u$ of structures, thus working implicitly with the instances $\mathsf{Structure}_\omega$ and Model_ω. This limitation is lifted in Section 8 by appealing to the downward Löwenheim–Skolem theorem.

Claessen et al. [10, § 2] define monotonicity on single sorts. Blanchette et al. [3, § 3] generalized the notion to sets of sorts S, making it more useful. The sorts S are collectively *monotonic* in the problem Φ if for all models $\mathcal{M}$ of Φ, there exists a model $\mathcal{M}'$ such that for all sorts σ, $\mathcal{M}'$ interprets σ by an infinite domain if $\sigma \in S$ and by a domain of the same cardinality as in $\mathcal{M}$ otherwise.

In the formalization, the Mono_Problem locale enriches Problem with a monotonicity assumption on all sorts, expressed using locale predicates:

$$\begin{array}{l} \big(\exists int_\mathsf{S}\ int_\mathsf{F}\ int_\mathsf{P}.\ \mathsf{Model}\ \mathsf{arity_F}\ \mathsf{res}\ \mathsf{arity_P}\ \Phi\ int_\mathsf{S}\ int_\mathsf{F}\ int_\mathsf{P}\big) \longrightarrow \\ \qquad \exists int_\mathsf{S}\ int_\mathsf{F}\ int_\mathsf{P}.\ \mathsf{Infinite\_Model}\ \mathsf{arity_F}\ \mathsf{res}\ \mathsf{arity_P}\ \Phi\ int_\mathsf{S}\ int_\mathsf{F}\ int_\mathsf{P} \end{array}$$

The Infinite_Model locale is itself an enrichment of Model with the assumption that for each sort σ, the expression $\mathsf{int_S}\ \sigma\ d$ is true for infinitely many elements d.

First Criterion. Claessen et al. designed two syntactic criteria to infer monotonicity. The first one is defined as a predicate $\rhd$ that checks the absence of naked variables of a given sort σ in a clause c or a problem Φ:

$$\sigma \rhd c \longleftrightarrow \forall x \in \mathsf{nv}\ c.\ \mathsf{sort}\ x \neq \sigma \qquad\qquad \sigma \rhd \Phi \longleftrightarrow \forall c \in \Phi.\ \sigma \rhd c$$

A *naked variable* is a variable that occurs directly on either side of a positive equality, such as X in the literal $X \approx \mathsf{f}(Y)$. Formally:

$$\begin{array}{lll} \mathsf{nv}\,(\mathsf{Var}\,x) = \{x\} & \mathsf{nv}\,(\mathsf{Eq}\,t_1\,t_2) = \mathsf{nv}\,t_1 \cup \mathsf{nv}\,t_2 & \mathsf{nv}\,(\mathsf{Pos}\,a) = \mathsf{nv}\,a \\ \mathsf{nv}\,(\mathsf{Fn}\,f\,ts) = \emptyset & \mathsf{nv}\,(\mathsf{Pr}\,p\,ts) = \emptyset & \mathsf{nv}\,(\mathsf{Neg}\,a) = \emptyset \end{array}$$

with $\mathsf{nv}\,c = \bigcup\,\mathsf{set}\,(\mathsf{map}\,\mathsf{nv}\,c)$ for clauses. The criterion $\rhd$ soundly infers monotonicity. This is expressed as a sublocale inclusion Problem_Crit1 < Mono_Problem, where Problem_Crit1 enriches Problem with the assumption $\forall\sigma.\ \sigma \rhd \Phi$. The inclusion holds because a model of a problem whose sorts pass $\rhd$ can be extended into an infinite model by replicating elements. For each finite sort σ, the extended model contains infinitely many copies of some element pick σ, all interpreted as in the original model.

Blanchette et al. strengthened the criterion by injecting “infinity knowledge”: Any sort that is interpreted by an infinite domain in all models is monotonic, regardless of naked variables [3, §3]. This aspect is part of the formalization but omitted here.

Second Criterion. The improved criterion is parameterized by an assignment of a per-sort *extension policy*—which may be *true*, *false*, or *copy*—to each predicate symbol. In the model construction, the true-extended (resp. false-extended) predicates are interpreted as true (resp. false) for new domain elements of the given sort, whereas the copy-extended predicates are treated as in the simple criterion.

Implementations can enumerate the possible policy combinations (e.g., using a SAT solver). In the formalization, the policies are supplied along with the problem as a curried function policy that maps pairs σ, p to T, F, or C. A function guard associates each variable x in need of protection with its guarding literal. The criterion is defined as

$$\begin{array}{l} \sigma \blacktriangleright c \longleftrightarrow \forall l\,x.\ l \in \mathsf{set}\,c \wedge x \in \mathsf{nv}\,l \wedge \mathsf{sort}\,x = \sigma \longrightarrow \mathsf{isGuard}\,x\,(\mathsf{guard}\,c\,l\,x) \\ \sigma \blacktriangleright \Phi \longleftrightarrow \forall c \in \Phi.\ \sigma \blacktriangleright c \end{array}$$

where isGuard determines whether the given literal actually protects the variable x:

$$\begin{array}{rcl} \mathsf{isGuard}\,x\,(\mathsf{Pos}\,(\mathsf{Eq}\,t_1\,t_2)) & \longleftrightarrow & \mathsf{False} \\ \mathsf{isGuard}\,x\,(\mathsf{Neg}\,(\mathsf{Eq}\,t_1\,t_2)) & \longleftrightarrow & \bigvee_{i=1}^{2} t_i = \mathsf{Var}\,x \wedge \exists f\,ts.\ t_{3-i} = \mathsf{Fn}\,f\,ts \\ \mathsf{isGuard}\,x\,(\mathsf{Pos}\,(\mathsf{Pr}\,p\,ts)) & \longleftrightarrow & x \in \bigcup\,\mathsf{set}\,(\mathsf{map}\,\mathsf{nv}\,ts) \wedge \mathsf{policy}\,(\mathsf{sort}\,x)\,p = \mathsf{T} \\ \mathsf{isGuard}\,x\,(\mathsf{Neg}\,(\mathsf{Pr}\,p\,ts)) & \longleftrightarrow & x \in \bigcup\,\mathsf{set}\,(\mathsf{map}\,\mathsf{nv}\,ts) \wedge \mathsf{policy}\,(\mathsf{sort}\,x)\,p = \mathsf{F} \end{array}$$

The notion of naked variables is generalized to account for ill-polarized predicates:

$$\begin{array}{rcl} \mathsf{nv}\,(\mathsf{Pos}\,(\mathsf{Pr}\,p\,ts)) & = & \{x \in \bigcup\,\mathsf{set}\,(\mathsf{map}\,\mathsf{nv}\,ts) \mid \mathsf{policy}\,(\mathsf{sort}\,x)\,p = \mathsf{F}\} \\ \mathsf{nv}\,(\mathsf{Neg}\,(\mathsf{Pr}\,p\,ts)) & = & \{x \in \bigcup\,\mathsf{set}\,(\mathsf{map}\,\mathsf{nv}\,ts) \mid \mathsf{policy}\,(\mathsf{sort}\,x)\,p = \mathsf{T}\} \end{array}$$

Theorem 1. *Let Φ be a Σ-problem and σ be a Σ-sort.*

(1) *If $\sigma \rhd \Phi$, then $\sigma \blacktriangleright \Phi$ for a copy-extended policy.*

(2) *Given some extension policies, if $\sigma \blacktriangleright \Phi$ for all Σ-sorts σ, then the set of all Σ-sorts is monotonic in Φ.*

This theorem is expressed in Isabelle as a pair of sublocale inclusions. The where clause below instantiates Problem_Policy_Crit2's policy parameter with $\lambda\sigma\,p.\ \mathsf{C}$ to enforce the copy policy for all sorts and predicate symbols:

```
sublocale Problem_Crit1 < Problem_Policy_Crit2 where policy = (λσ p. C)
sublocale Problem_Policy_Crit2 < Mono_Problem
```

5 Sort Encodings

A naive, unsound way to translate a many-sorted FOL problem to unsorted FOL is to erase all the sorts and otherwise leave the problem unchanged. There are two main sound alternatives that encode the sort information. Sort *tags* are functions $\mathsf{t}_\sigma(X)$ that directly associate a term X with its sort σ. Sort *guards* are predicates $\mathsf{g}_\sigma(X)$ that check whether X has sort σ in the original problem. The formalized versions of these encodings follow the four steps sketched in Section 2.

Full Erasure. Full sort erasure is unsound but complete. What makes it interesting is that it is sound for the class of monotonic problems. By way of composition, it lies at the heart of the tag- and guard-based encodings. The theory prefix U distinguishes unsorted entities from their many-sorted counterparts.

SIG: The signature of the target unsorted problem has the same function and predicate symbols as the original signature but collapses the sorts into a single, implicit sort.

TRANS: The translation function e is the identity except that it forgets the sorts.

SOUND: For the soundness proof, a model of a monotonic problem is extended into a model that interprets all sorts infinitely, which in turn is transformed into an isomorphic "full" model that interprets all the sorts uniformly as $\lambda d.\,\mathsf{True}$ (i.e., $\forall\sigma.\ \forall d.\ \mathsf{int}_\mathsf{S}\ \sigma\ d$), from which it is easy to build an unsorted model for the e-translated problem:

$$\mathsf{Mono\_Model} < \mathsf{Infinite\_Model} < \mathsf{Full\_Model} < \mathsf{U.Model}$$

The last step corresponds to Theorem 1 in Bouillaguet et al. [9] and, more approximately, to Lemma 1 in Claessen et al. [10]. Incidentally, the formalization revealed a flaw in Claessen et al.: Their main result holds, but not their Lemma 3.[1]

COMPLETE: The locale Problem_UModel combines a many-sorted problem and an unsorted model with domain D of the problem's e translation. The unsorted model can be regarded as a many-sorted model in which every sort is interpreted as D.

Protector-Based Encodings. Claessen et al. observe that protectors, whether tags or guards, are not needed for terms with monotonic sorts. The sequel [3] advocates protecting only those variables that cause the monotonicity check to fail, to reduce clutter. Thus, for both tags and guards, three schemes are available: the traditional encoding, the lightweight version due to Claessen et al., and the "featherweight" version from the sequel. These are called $\widetilde{\mathsf{t}}$, $\widetilde{\mathsf{t}}?$, and $\widetilde{\mathsf{t}}??$ for tags and $\widetilde{\mathsf{g}}$, $\widetilde{\mathsf{g}}?$, and $\widetilde{\mathsf{g}}??$ for guards.

Consider the following fragment of a many-sorted problem, where S has sort st:

$$S \approx \mathsf{on} \lor S \approx \mathsf{off} \qquad\qquad \mathsf{flip}(S) \not\approx S$$

[1] The flawed lemma states that whenever there exists a model $\mathcal{M}$ where a monotonic sort σ is interpreted with a given cardinality, there exists for any larger cardinality k a model where σ has cardinality k and the other sorts have the same cardinalities as in $\mathcal{M}$. This proposition is invalid for $k > \aleph_0$ because FOL problems can encode the constraint that there exists a bijection between two infinite, and hence monotonic, sorts σ and τ, making it impossible to increase σ's cardinality without also increasing τ's. This issue is independent of which of the two definitions of monotonicity is used. We discovered it at an early stage of the formalization as we were looking for a correct formulation of Löwenheim–Skolem for many-sorted FOL.

The traditional $\widetilde{\mathsf{t}}$ encoding inserts tags around every subterm:

$$\mathsf{t}_{\mathsf{st}}(S) \approx \mathsf{t}_{\mathsf{st}}(\mathsf{on}) \lor \mathsf{t}_{\mathsf{st}}(S) \approx \mathsf{t}_{\mathsf{st}}(\mathsf{off}) \qquad \mathsf{t}_{\mathsf{st}}(\mathsf{flip}(\mathsf{t}_{\mathsf{st}}(S))) \not\approx \mathsf{t}_{\mathsf{st}}(S)$$

Since the sort st is not monotonic (its only models have cardinality 2), the $\widetilde{\mathsf{t}}?$ encoding coincides with $\widetilde{\mathsf{t}}$. In contrast, the featherweight $\widetilde{\mathsf{t}}??$ encoding tags only naked variables:

$$\mathsf{t}_{\mathsf{st}}(S) \approx \mathsf{on} \lor \mathsf{t}_{\mathsf{st}}(S) \approx \mathsf{off} \qquad \mathsf{flip}(S) \not\approx S$$

The $\widetilde{\mathsf{t}}??$-encoded problem is complemented by typing axioms that repair mismatches between tagged and untagged occurrences of well-sorted terms:

$$\mathsf{t}_{\mathsf{st}}(\mathsf{on}) \approx \mathsf{on} \qquad \mathsf{t}_{\mathsf{st}}(\mathsf{off}) \approx \mathsf{off} \qquad \mathsf{t}_{\mathsf{st}}(\mathsf{flip}(S)) \approx \mathsf{flip}(S)$$

For guards, the traditional and lightweight encodings $\widetilde{\mathsf{g}}$ and $\widetilde{\mathsf{g}}?$ protect each variable:

$$\neg\, \mathsf{g}_{\mathsf{st}}(S) \lor S \approx \mathsf{on} \lor S \approx \mathsf{off} \qquad \neg\, \mathsf{g}_{\mathsf{st}}(S) \lor \mathsf{flip}(S) \not\approx S$$

The featherweight encoding $\widetilde{\mathsf{g}}??$ guards only naked variables:

$$\neg\, \mathsf{g}_{\mathsf{st}}(S) \lor S \approx \mathsf{on} \lor S \approx \mathsf{off} \qquad \mathsf{flip}(S) \not\approx S$$

The guard encodings are completed by the axioms $\mathsf{g}_{\mathsf{st}}(\mathsf{on})$, $\mathsf{g}_{\mathsf{st}}(\mathsf{off})$, and $\mathsf{g}_{\mathsf{st}}(\mathsf{flip}(S))$.

General Encoding Procedure. The full sort erasure encoding e is part of a two-stage procedure to encode any many-sorted FOL problem into unsorted FOL. The first stage makes the problem monotonic by introducing protectors (tags or guards). This corresponds to a sound and complete encoding of many-sorted FOL into itself; the soundness proofs rely on the monotonicity criteria. The second stage merges all the sorts using e, which is sound and complete for monotonic problems.

Tags and guards are formalized separately, but for a protector kind, the traditional, lightweight, and featherweight encodings are treated as instances of a single generalized encoding. Both generalized encodings are parameterized by a partition of sorts by level of protection, via disjoint predicates prot, protFw, unprot : $'s \to$ bool indicating whether terms of a sort should be fully protected, protected in a featherweight fashion, or left unprotected. The last option is available only for sorts inferred monotonic by $\rhd$.

Tags. The tag encoding builds on a datatype of extended function symbols containing the old symbols as well as a tag for each sort:

$\texttt{datatype}\ ('f, 's)\ \mathsf{efsym} = \mathsf{Old}\ 'f \mid \mathsf{Tag}\ 's$

SIG: Signatures over the extended symbols treat the old function symbols as before. The new symbols Tag σ are unary operations of sort arity $[\sigma]$ and result sort σ.

TRANS: The encoding function is specified as follows:

$$\begin{aligned}
\mathsf{t}\ (\mathsf{Var}\ x) &= \begin{cases} \mathsf{Var}\ x & \text{if } \mathsf{unprot}\ (\mathsf{sort}\ x) \\ \mathsf{Fn}\ (\mathsf{Tag}\ (\mathsf{sort}\ x))\ [\mathsf{Var}\ x] & \text{otherwise} \end{cases} \\
\mathsf{t}\ (\mathsf{Fn}\ f\ ts) &= \mathsf{t}'\ (\mathsf{Fn}\ f\ ts) \\
\mathsf{t}\ (\mathsf{Pos}\ (\mathsf{Eq}\ t_1\ t_2)) &= \mathsf{Pos}\ (\mathsf{Eq}\ (\mathsf{t}\ t_1)\ (\mathsf{t}\ t_2)) \\
\mathsf{t}\ (\mathsf{Neg}\ (\mathsf{Eq}\ t_1\ t_2)) &= \mathsf{Neg}\ (\mathsf{Eq}\ (\mathsf{t}'\ t_1)\ (\mathsf{t}'\ t_2)) \\
\mathsf{t}\ (\mathsf{Pos}\ (\mathsf{Pr}\ p\ ts)) &= \mathsf{Pos}\ (\mathsf{Pr}\ p\ (\mathsf{map}\ \mathsf{t}'\ ts)) \\
\mathsf{t}\ (\mathsf{Neg}\ (\mathsf{Pr}\ p\ ts)) &= \mathsf{Neg}\ (\mathsf{Pr}\ p\ (\mathsf{map}\ \mathsf{t}'\ ts))
\end{aligned}$$

$$\mathsf{t}'\ (\mathsf{Var}\ x) = \begin{cases} \mathsf{Fn}\ (\mathsf{Tag}\ (\mathsf{sort}\ x))\ [\mathsf{Var}\ x] & \text{if } \mathsf{prot}\ (\mathsf{sort}\ x) \\ \mathsf{Var}\ x & \text{otherwise} \end{cases}$$

$$\mathsf{t}'\ (\mathsf{Fn}\ f\ ts) = \begin{cases} \mathsf{Fn}\ (\mathsf{Tag}\ (\mathsf{res}\ f))\ [\mathsf{Fn}\ (\mathsf{Old}\ f)\ (\mathsf{map}\ \mathsf{t}'\ ts)] & \text{if } \mathsf{prot}\ (\mathsf{res}\ f) \\ \mathsf{Fn}\ (\mathsf{Old}\ f)\ (\mathsf{map}\ \mathsf{t}'\ ts) & \text{otherwise} \end{cases}$$

The t function tags naked variables unless they are of an unprotected sort. The auxiliary function t' adds tags only for fully protected sorts; it is invoked on all subterms except naked variables.

The tag axioms $\mathcal{A}x_\Phi$—needed to repair mismatches between tagged and untagged terms in the featherweight encoding $\widetilde{\mathsf{t}}$??—have the form $\mathsf{Pos}\,(\mathsf{Eq}\,(\mathsf{Fn}\,(\mathsf{Tag}\,(\mathsf{res}\,f)\,[t]))\,t)$, where $t = \mathsf{Fn}\ (\mathsf{Old}\ f)\ (\mathsf{map}\ \mathsf{Var}\ xs)$ and xs is a list of distinct variables of sorts $\mathsf{arity_F}\ f$, for all function symbols f such that $\mathsf{protFw}\ (\mathsf{res}\ f)$. The encoding of a problem is $\mathsf{t}\ \Phi = \{\mathsf{map}\ \mathsf{t}\ c \mid c \in \Phi\} \cup \mathcal{A}x_\Phi$.

SOUND: Given a model of the fixed problem Φ, a model of $\mathsf{t}\ \Phi$ is obtained by extending it with interpreting tags as the identity functions.

COMPLETE: Completeness is more difficult. To convey a sense of the complexity, let us quote the informal proof, in which x stands for $\widetilde{\mathsf{t}}$? or $\widetilde{\mathsf{t}}$?? ($\widetilde{\mathsf{t}}$ is analogous to $\widetilde{\mathsf{t}}$?) and $[\![\Phi]\!]_{\mathsf{x}}$ denotes the x-encoding of the NNF problem Φ [4, §4.4]:

> A model of $[\![\Phi]\!]_{\mathsf{x}}$ is *canonical* if all tag functions t_σ are interpreted as the identity. From a canonical model, we obtain a model of Φ by leaving out t_σ. It then suffices to prove that whenever there exists a model $\mathcal{M}$ of $[\![\Phi]\!]_{\mathsf{x}}$, there exists a canonical model $\mathcal{M}'$.
>
> For $\widetilde{\mathsf{t}}$?, values of a tagged type σ are systematically accessed through t_σ. Hence, we can safely permute the entries of the function table of each t_σ so that it is the identity for the values in its range. We then construct $\mathcal{M}'$ by removing the domain elements for which t_σ is not the identity. It is a model by Lemma 4.13 [which states that substructures of NNF models are models if they preserve existential witnesses].
>
> For $\widetilde{\mathsf{t}}$??, the construction must take possibly nonmonotonic types into account. No permutation is necessary for these thanks to the typing axioms, which ensure that the tag functions are the identity for well-typed terms. For each $\sigma \not\rhd \Phi$, we remove the model elements for which t_σ is not the identity. The typing axioms ensure that the substructure is well-defined: each tag function is the identity for at least one element and also for each element within the range of a non-tag function. The equations $\mathsf{t}_\sigma(X) \approx X$ generated for existential variables ensure that witnesses are preserved, as required by Lemma 4.13.

Relying on permutations is intuitive on paper, but in the proof assistant it is simpler to combine the permutation and the reduction to a canonical model:

$$\mathsf{int_F}\ f\ as = \begin{cases} \mathsf{eint_F}\ (\mathsf{Tag}\,(\mathsf{res}\,f))\ [\mathsf{eint_F}\ (\mathsf{Old}\ f)\ (\mathsf{map_2}\ \mathsf{q}\ (\mathsf{arity_F}\ f)\ as)] & \text{if } \mathsf{prot}\,(\mathsf{res}\ f) \\ \mathsf{eint_F}\ (\mathsf{Old}\ f)\ (\mathsf{map_2}\ \mathsf{q}\ (\mathsf{arity_F}\ f)\ as) & \text{otherwise} \end{cases}$$

Here, $\mathsf{eint_F}$ denotes the $\mathsf{int_F}$ component of the fixed model of $\mathsf{t}\ \Phi$, and $\mathsf{map_2}$ applies a binary function elementwise on parallel lists. The auxiliary function q maps a sort σ

and an element d to d if either $\mathsf{unprot}\ \sigma$ or d is in the range of $\mathsf{eint}_\mathsf{F}\ (\mathsf{Tag}\ \sigma)$; otherwise, it maps σ, d to $\mathsf{eint}_\mathsf{F}\ (\mathsf{Tag}\ \sigma)\ d$. The proof that the resulting structure is a model of the original problem Φ involves defining suitable back-and-forth functions between the two structures. Finally, proving monotonicity of $\mathsf{t}\ \Phi$ is reduced to showing that the first criterion always succeeds on the translated problem: $\mathsf{Problem}_\Phi < \mathsf{Problem\_Crit1}_{\mathsf{t}\,\Phi}$.

Guards. The guard encoding requires extending the signature with guard predicates:

```
datatype ('p,'s) epsym = Old 'p | Guard 's
```

Each symbol $\mathsf{Guard}\ \sigma$ has arity $[\sigma]$ and contributes axioms to the translated problem.

The soundness proof extends models of Φ into models of $\mathsf{g}\ \Phi$ by interpreting the guard predicates as true everywhere. The completeness part is easier for guards than for tags. A canonical model is one where all guard predicates are interpreted as true everywhere. The proof handles the three levels of protection uniformly, reflecting the more uniform nature of $\widetilde{\mathsf{g}}$??—there are no counterparts to the "typing axioms that repair mismatches between tagged and untagged occurrences of well-sorted terms" of $\widetilde{\mathsf{t}}$??.

Monotonicity is proved using the second criterion, with the extension policy C for the predicates $\mathsf{Old}\ p$ and F for the distinguished symbols $\mathsf{Guard}\ \sigma$. This is a departure from the informal proof, which inlines the model extension argument without appealing to the monotonicity criterion.

6 First-Order Logic with Quantifiers

This and the next two sections are concerned with lifting the results presented in the previous sections to negation normal form and structures with arbitrarily large domains.

The locales for quantified FOL formulas in NNF are either the same or similar to those for CNF; the theory prefix Q is used for disambiguation (e.g., $\mathsf{Q.Model}$). No cardinality assumption is made about the universe. Terms and atoms are as for CNF, but formulas can nest positive connectives and quantifiers arbitrarily.

The following declaration gives an approximation of the syntactic category of formulas. The actual type identifies them modulo α-equivalence (variable renaming):

```
datatype ('s,'f,'p) fm =
  Pos (('f,'p) atm) | Conj (('s,'f,'p) fm) (('s,'f,'p) fm) | All 's var (('s,'f,'p) fm) |
  Neg (('f,'p) atm) | Disj (('s,'f,'p) fm) (('s,'f,'p) fm)  | Ex 's var (('s,'f,'p) fm)
```

The proper formal management of binding syntax modulo α-equivalence is a topic of extensive research in λ-calculus and programming languages. FOL syntax poses similar challenges. In particular, substitution and its interplay with the semantics is difficult to handle rigorously; for example, a standard textbook [16] dedicates dozens of lemmas to these preliminaries, with rough proof sketches. Many of these refer to properties of any syntax with static bindings, falling under the scope of a general metatheory of syntax formalized by Popescu et al. [23, 24]. A prominent feature of this framework—distinguishing it from the more established Nominal Isabelle [14], based on nominal logic [21]—is that it is centered around the notion of substitution:

- The framework defines substitution, including parallel and unary variants, and provides a large collection of basic facts about the interaction of substitution with free variables and the other operators.

- It provides a recursor for defining operators that are directly compositional with substitution. (In contrast, the nominal logic recursor targets compositionality with permutations, a less useful concept.)

This unconventional focus is appropriate: Substitution is without doubt the central syntactic operator in logics and type systems.

Another main feature is the facilitation of semantic interpretation of syntax, which is problematic in frameworks optimized for manipulating finitary syntax. For example, Pitts encounters "a really nontrivial freshness condition on binders" [22, § 6.3] he needs to discharge in the context of applying the nominal recursor to interpret the λ-calculus in a semantic domain. This feature is illustrated below for interpreting FOL syntax.

The framework requires the user to provide semantic domains—for FOL, types $\mathcal{T}$, $\mathcal{A}$, and $\mathcal{F}$ for interpreting terms, atoms, and formulas—as well as first-order operations corresponding to the non-binding constructors other than for variables (e.g., $\mathsf{FN} : {'f} \to \mathcal{T}\ \mathsf{list} \to \mathcal{T}$) and second-order operations corresponding to the binders: $\mathsf{ALL} : {'s} \to (\mathcal{T} \to \mathcal{F}) \to \mathcal{F}$ and $\mathsf{EX} : {'s} \to (\mathcal{T} \to \mathcal{F}) \to \mathcal{F}$.

In exchange, the framework produces the functions $\mathsf{int}_{\mathsf{Tm}} : \mathsf{tm} \to (\mathsf{var} \to \mathcal{T}) \to \mathcal{T}$, $\mathsf{int}_{\mathsf{At}} : \mathsf{atm} \to (\mathsf{var} \to \mathcal{T}) \to \mathcal{A}$, and $\mathsf{int}_{\mathsf{Fm}} : \mathsf{fm} \to (\mathsf{var} \to \mathcal{T}) \to \mathcal{F}$ that interpret syntax in the semantic domains. They map variables according to a valuation ξ. They map the action of non-binding constructors to that of the corresponding semantic operators, and similarly for binding constructors but in a valuation-sensitive way. For example:

$$\begin{aligned}
\mathsf{int}_{\mathsf{Tm}}\ (\mathsf{Var}\ x)\ \xi &= \xi\ x \\
\mathsf{int}_{\mathsf{Tm}}\ (\mathsf{Fn}\ f\ ts)\ \xi &= \mathsf{FN}\ f\ (\mathsf{map}\ (\lambda t.\ \mathsf{int}_{\mathsf{Tm}}\ t\ \xi)\ ts) \\
\mathsf{int}_{\mathsf{At}}\ (\mathsf{Eq}\ t_1\ t_2)\ \xi &= \mathsf{EQ}\ (\mathsf{int}_{\mathsf{Tm}}\ t_1\ \xi)\ (\mathsf{int}_{\mathsf{Tm}}\ t_2\ \xi) \\
\mathsf{int}_{\mathsf{Fm}}\ (\mathsf{Conj}\ \varphi_1\ \varphi_2)\ \xi &= \mathsf{CONJ}\ (\mathsf{int}_{\mathsf{Fm}}\ \varphi_1\ \xi)\ (\mathsf{int}_{\mathsf{Fm}}\ \varphi_2\ \xi) \\
\mathsf{int}_{\mathsf{Fm}}\ (\mathsf{All}\ \sigma\ x\ \varphi)\ \xi &= \mathsf{ALL}\ \sigma\ (\lambda d.\ \mathsf{int}_{\mathsf{Fm}}\ \varphi\ \xi[x \mapsto d])
\end{aligned}$$

where $\xi[x \mapsto d]$ denotes the function that maps x to d and otherwise coincides with ξ. So far, this looks like the standard interpretation of binding syntax in a semantic domain, except that here the recursive definition is modulo α-equivalence (which is a priori difficult to achieve in a proof assistant). The framework also derives compositionality of substitution w.r.t. valuation update and obliviousness of the interpretation w.r.t. fresh variables in a systematic, FOL-agnostic way:

$$\begin{aligned}
\mathsf{int}_{\mathsf{Fm}}\ \varphi[t/x]\ \xi &= \mathsf{int}_{\mathsf{Fm}}\ \varphi\ \xi[x \mapsto \mathsf{int}_{\mathsf{Tm}}\ t\ \xi] \\
\mathsf{int}_{\mathsf{Fm}}\ \varphi\ \xi &= \mathsf{int}_{\mathsf{Fm}}\ \varphi\ \xi' \qquad \text{if } \xi \text{ and } \xi' \text{ differ only on variables fresh for } \varphi
\end{aligned}$$

In the first equation, $\varphi[t/x]$ denotes capture-free substitution of t for x in φ.

A many-sorted structure $(\mathsf{int}_{\mathsf{S}}, \mathsf{int}_{\mathsf{F}}, \mathsf{int}_{\mathsf{P}})$ can be organized as a semantic domain by taking $\mathcal{T} = \omega$, $\mathcal{A} = \mathcal{F} = \mathsf{bool}$, $\mathsf{FN} = \mathsf{int}_{\mathsf{F}}$, $\mathsf{EQ} = (=)$, $\mathsf{CONJ} = (\wedge)$, $\mathsf{ALL}\ \sigma\ P = (\forall d.\ \mathsf{int}_{\mathsf{S}}\ \sigma\ d \longrightarrow P\ d)$, and so on. This yields the recursive equations

$$\begin{aligned}
[\![\mathsf{Var}\ x]\!]_\xi &= \xi\ x \\
[\![\mathsf{Fn}\ f\ ts]\!]_\xi &= \mathsf{int}_{\mathsf{F}}\ f\ (\mathsf{map}\ (\lambda t.\ [\![t]\!]_\xi)\ ts) \\
\models_\xi \mathsf{Eq}\ t_1\ t_2 &\longleftrightarrow [\![t_1]\!]_\xi = [\![t_2]\!]_\xi \\
\models_\xi \mathsf{Conj}\ \varphi_1\ \varphi_2 &\longleftrightarrow \models_\xi \varphi_1 \wedge \models_\xi \varphi_2 \\
\models_\xi \mathsf{All}\ \sigma\ x\ \varphi &\longleftrightarrow \forall d.\ \mathsf{int}_{\mathsf{S}}\ \sigma\ d \longrightarrow \models_{\xi[x \mapsto d]} \varphi
\end{aligned}$$

which characterize term interpretation (with $[\![t]\!]_\xi = \mathsf{int}_{\mathsf{Tm}}\, t\, \xi$), atom satisfaction ($\models_\xi a = \mathsf{int}_{\mathsf{At}}\, a\, \xi$), and formula satisfaction ($\models_\xi \varphi = \mathsf{int}_{\mathsf{Fm}}\, \varphi\, \xi$). These functions are defined in the Q.Structure locale. The framework also produces the substitution lemma $\models_\xi \varphi[t/x] \longleftrightarrow \models_{\xi[x \mapsto [\![t]\!]_\xi]} \varphi$. In the next section, the notations $\models \varphi$ and $\models \Phi$ abbreviate $\forall \xi.\ \models_\xi \varphi$ and $\forall \varphi \in \Phi.\ \models \varphi$. The structure can also be made explicit—e.g., $(\mathsf{int}_{\mathsf{S}}, \mathsf{int}_{\mathsf{F}}, \mathsf{int}_{\mathsf{P}}) \models_\xi \varphi$.

If the orientation toward substitution is the main strength of the framework, its main weakness is the lack of automation. For each desired binding syntax type, users must currently instantiate the general theorems manually, much like mathematicians do routinely when applying universal algebra to groups or rings. The instantiation is tedious due to the large number of theorems. Despite the availability of “template files,” this process can take days and thousands of lines of proof text. Automation in the form of a definitional package, which would provide the basic convenience expected by users of Nominal Isabelle (while supporting substitution natively), remains for future work.

7 Classical Metatheorems

The lifting argument from countable CNF structures to unbounded NNF structures (Section 8) relies on clausification and Löwenheim–Skolem for many-sorted FOL with equality. Earlier formalizations focus on unsorted FOL without equality [2,12,25]. Sorts and equality are tedious to formalize, and they often fail to reward the formalizer with deep logical insight, but they are central to monotonicity and sort encodings.

Clausification. The translation of a finite quantified problem into clausal form involves skolemizing all the existentially quantified variables into function symbols that take the universally quantified variables in scope as arguments. Skolemization is surprisingly difficult to treat formally; for example, Harrison [12] claims that it poses greater challenges than completeness. On the positive side, clausification can be seen as an instance of the general semantic interpretation principle introduced in Section 6.

The definition of clausification and its soundness and completeness proof follow the four-step institutional approach.

SIG: Skolemization introduces new function symbols $\mathsf{Sko}\ \sigma s\ x$, built from a list of sorts σs (specifying the arity) and a variable name x, while preserving the sorts of Σ-symbols:

$$\mathtt{datatype}\ 'f\ \mathsf{efsym}\ =\ \mathsf{Old}\ 'f\ \mid\ \mathsf{Sko}\ ('s\ \mathsf{list})\ \mathsf{var}$$

TRANS: The clausification function cls takes a Σ-formula φ, an environment $\rho : \mathsf{var} \to \mathsf{tm}$, a list of universal variables vs, and a set of fresh variables V as arguments. In addition to massaging the connectives, it replaces existential variables by new symbols that depend on vs, replaces bound universal variables by fresh variables from V, and substitutes free variables according to ρ to produce a Σ'-clause.

The characteristic equations for cls are obtained by instantiating the semantic interpretation principle with $\mathcal{T} = \mathsf{tm}$, $\mathcal{A} = \mathsf{atm}$, and $\mathcal{F} = \mathsf{var\ list} \to \mathsf{var\ set} \to \mathsf{fm}$, taking suitable operators on these domains, and letting cls be $\mathsf{int}_{\mathsf{Fm}}$. The interesting cases are

$$\begin{aligned} \mathsf{cls}\ (\mathsf{All}\ \sigma\ x\ \varphi)\ \rho\ vs\ V &= \mathsf{cls}\ \varphi\ \rho[x \mapsto \mathsf{Var}\ v]\ (v \# vs)\ (V \setminus \{v\}) \\ \mathsf{cls}\ (\mathsf{Ex}\ \sigma\ x\ \varphi)\ \rho\ vs\ V &= \mathsf{cls}\ \varphi\ \rho[x \mapsto \mathsf{Fn}\ f\ (\mathsf{map\ Var}\ vs)]\ vs\ (V \setminus \{v\}) \end{aligned}$$

where $v \in V$ is some variable of sort σ and $f = \mathsf{Sko}\ (\mathsf{map\ sort}\ vs)\ v$ is the Skolem function symbol, which is applied to the universal variables vs. For closed formulas, clausification is defined as $\mathsf{clausify}\ \varphi = \mathsf{cls}\ \varphi\ \rho\ []\ \mathsf{UNIV}$ for some irrelevant choice of ρ.

As a simple example, let $\varphi = \mathsf{All}\ \sigma\ x\ (\mathsf{Ex}\ \tau\ y\ (\mathsf{Eq}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ y)))$, let v_1, v_2 be the variables picked from UNIV and $\mathsf{UNIV} \setminus \{v_1\}$, and let $f = \mathsf{Sko}\ [\sigma]\ v_2$. Then

$$\begin{aligned}
&\mathsf{clausify}\ \varphi \\
&= \mathsf{cls}\ \varphi\ \rho\ []\ \mathsf{UNIV} \\
&= \mathsf{cls}\ (\mathsf{Ex}\ \tau\ y\ (\mathsf{Eq}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ y)))\ \rho[x \mapsto \mathsf{Var}\ v_1]\ [v_1]\ (\mathsf{UNIV} \setminus \{v_1\}) \\
&= \mathsf{cls}\ (\mathsf{Eq}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ y))\ \rho[x \mapsto \mathsf{Var}\ v_1,\ y \mapsto \mathsf{Fn}\ f\ [\mathsf{Var}\ v_1]]\ [v_1]\ (\mathsf{UNIV} \setminus \{v_1, v_2\}) \\
&= \mathsf{Eq}\ (\mathsf{Var}\ v_1)\ (\mathsf{Fn}\ f\ [\mathsf{Var}\ v_1])
\end{aligned}$$

SOUND: Soundness is proved in the Structure locale, which fixes a Σ-structure $(\mathsf{int_S}, \mathsf{int_F}, \mathsf{int_P})$. The "Skolem model" predicate $\mathsf{skmod}\ \varphi\ \rho\ vs\ V\ eint_{\mathsf{F}}\ eint'_{\mathsf{F}}$ transforms, for each valuation $\xi : \mathsf{var} \to {}'u$, an extended structure $eint_{\mathsf{F}}$ such that $\models_\xi \mathsf{cls}\ \varphi\ \rho\ vs\ V$ into an extended structure $eint'_{\mathsf{F}}$ such that $\models_{\xi \diamond \rho} \varphi$, where $\diamond$ composes valuations with environments. The introduction rules of skmod emulate cls's equations; for example,

$$\frac{\mathsf{skmod}\ \varphi\ \rho[x \mapsto \mathsf{Fn}\ f\ (\mathsf{map\ Var}\ vs)]\ vs\ (V \setminus \{v\})\ eint_{\mathsf{F}}[f \mapsto F]\ eint'_{\mathsf{F}}}{\mathsf{skmod}\ (\mathsf{Ex}\ \sigma\ x\ \varphi)\ \rho\ vs\ V\ eint_{\mathsf{F}}\ eint'_{\mathsf{F}}}$$

where $v \in V$ and $F : {}'u\ \mathsf{list} \to {}'u$ is a suitable interpretation for the Skolem symbol f, defined so that $F\ us$ gives an arbitrary u such that $(\mathsf{int_S}, \mathsf{int_F}, \mathsf{int_P}) \models_{\xi \diamond \rho[x \mapsto u]} \varphi$, where ξ maps vs to us elementwise. The skmod relation is total on the last argument. For closed formulas φ such that $(\mathsf{int_S},\mathsf{int_F},\mathsf{int_P}) \models \varphi$, starting with an extension $\mathsf{eint_F}$ of $\mathsf{int_F}$, skmod yields $\mathsf{eint'_F}$ such that $(\mathsf{int_S},\mathsf{eint'_F},\mathsf{int_P}) \models \mathsf{clausify}\ \varphi$. Thus, if φ has a model, then $\mathsf{clausify}\ \varphi$ also has a model.

For problems, we define $\mathsf{clausify}\ \Phi = \mathsf{clausify}\ (\bigwedge\Phi)$, where $\bigwedge\Phi$ is the conjunction of all formulas in Φ, which must be finite. The locale Q.Model fixes Φ and a model, which is also a model of the formula $\bigwedge\Phi$. By soundness of clausify on closed formulas, this yields a model of $\mathsf{clausify}\ \Phi$.

COMPLETE: For completeness, it suffices to show that the backward structure translation of a model of $\mathsf{clausify}\ \Phi$ is a model of Φ. This is straightforward.

Löwenheim–Skolem. The proof of the downward Löwenheim–Skolem theorem is based on a formalization of a complete inference system, described in a separate paper [8]. In the Q.Model locale, which fixes a problem and model, it constructs a syntactic Henkin model. Since this model has a countable universe, there exists an isomorphic copy on ω (the countably infinite universe fixed throughout Sections 4 and 5). This yields $\mathsf{Q.Model}_{'u} < \mathsf{Q.Model}_{\omega}$.

Using the obvious sound and complete embedding embed of CNF problems into NNF problems, it is possible to transfer the Löwenheim–Skolem theorem to CNF:

$$\mathsf{Model}_{'u,\Phi} < \mathsf{Q.Model}_{'u,\mathsf{embed}\,\Phi} < \mathsf{Q.Model}_{\omega,\mathsf{embed}\,\Phi} < \mathsf{Model}_{\omega,\Phi}$$

To summarize the results of this section:

Theorem 2. *An NNF problem Φ has a model iff* clausify Φ *has a model.*

Theorem 3. *An NNF problem has a model iff it has a countable model.*

8 Lifting to Arbitrary Structures and Formulas with Binders

The focus on clausal form and countable structures is a useful simplification, but it is not faithful to the NNF-based paper proof [3] (or to the implementation in Sledgehammer). Thanks to a lifting argument that relies on clausification and Löwenheim–Skolem, the final results are free of such restrictions.

Figure 1 shows how the results are connected. Starting at the top with a satisfiable quantified problem Φ, the problem is first clausified, then by Löwenheim–Skolem it is countably satisfiable (by taking $'u = \omega$). On the left-hand side, the clausified problem is further encoded using tags or guards ($x \in \{\mathsf{t}, \mathsf{g}\}$) and shown to pass one of the monotonicity criteria ($i = 1$ for t and 2 for g), meaning it is monotonic. On the right-hand side, the encoded problem is satisfiable. Merging the two branches yields a monotonic satisfiable problem, whose erasure is a satisfiable unsorted problem. Since every translation step is also shown complete, the right-hand side can also be traversed bottom-up, producing a model of the original problem from a model of the translated unsorted problem. The overall translation is thus sound and complete.

Theorem 4. *Given $x \in \{\mathsf{t}, \mathsf{g}\}$ and a finite many-sorted NNF problem Φ, let Φ' be the unsorted CNF problem $\mathsf{e}\,(x\,(\mathsf{clausify}\,\Phi))$, i.e., the sort-erased x-translated clausified Φ.*

(1) *For each model $\mathcal{M}$ of Φ (forming together with Φ an instance of $\mathsf{Q.Model}_{\Phi}$), there exists a model $\mathcal{M}'$ of Φ' (forming together with Φ' an instance of $\mathsf{U.Model}_{\Phi'}$).*

(2) *Conversely, for every model of Φ', there exists a model of Φ.*

The formal proof puts together many constructions and results of independent interest, notably soundness of the monotonicity criteria (Theorem 1), soundness and completeness of clausification (Theorem 2), and downward Löwenheim–Skolem (Theorem 3).

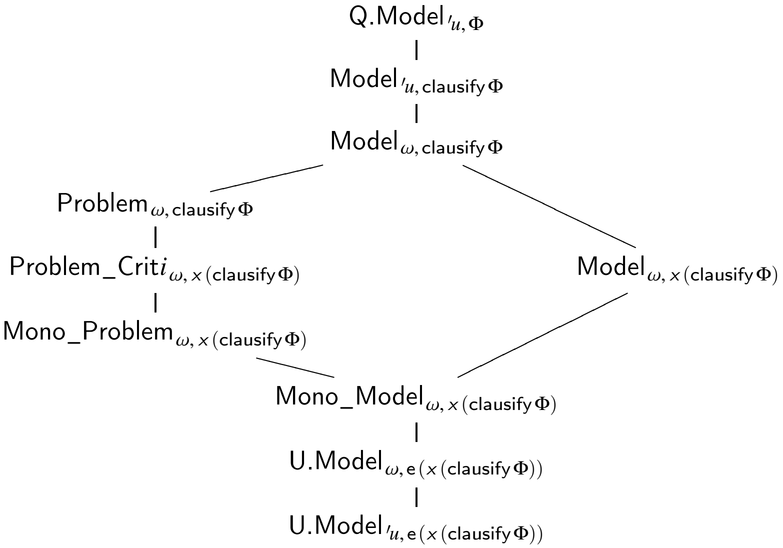

$\mathsf{Q.Model}_{'u,\,\Phi}$
$\mathsf{Model}_{'u,\,\mathsf{clausify}\,\Phi}$
$\mathsf{Model}_{\omega,\,\mathsf{clausify}\,\Phi}$
$\mathsf{Problem}_{\omega,\,\mathsf{clausify}\,\Phi}$
$\mathsf{Problem\_Crit}i_{\omega,\,x\,(\mathsf{clausify}\,\Phi)}$
$\mathsf{Mono\_Problem}_{\omega,\,x\,(\mathsf{clausify}\,\Phi)}$
$\mathsf{Model}_{\omega,\,x\,(\mathsf{clausify}\,\Phi)}$
$\mathsf{Mono\_Model}_{\omega,\,x\,(\mathsf{clausify}\,\Phi)}$
$\mathsf{U.Model}_{\omega,\,\mathsf{e}\,(x\,(\mathsf{clausify}\,\Phi))}$
$\mathsf{U.Model}_{'u,\,\mathsf{e}\,(x\,(\mathsf{clausify}\,\Phi))}$

Fig. 1. The verified translation pipeline

9 Conclusion

This paper describes a framework and a methodology for formalizing applications of many-sorted first-order logic while acting as a companion to recent papers on sort encodings [3, 9, 10]. To readers from the proof assistant community, it also provides a contribution to the ongoing binder representation debate. And to readers rooted in algebraic methods, it shows a practical application of the theory of institutions in a context where the translation functions cannot be assumed to be uniform.

The formalization widely reaffirmed already proved results. On one occasion, it revealed a flaw in a published lemma (Lemma 3 of Claessen et al. [10]). It also helped detect mistakes in a subsequent paper proof [4] before it reached any readers. The work provided the opportunity to rethink the proof; for example, the generalized monotonicity concept, in terms of sets of sorts, arose during the formalization.

A potential practical benefit of this work is connected to step-by-step proof reconstruction. Although the encodings are sound, the inferences in a machine-generated proof may violate the sort discipline, resulting in failures in Sledgehammer's proof replay. In future work, we want to investigate the feasibility of connecting the soundness proofs of the encodings with a verified checker for unsorted FOL proofs.

The advantages of machine-checked metatheory are well known from programming language research, where papers are often accompanied by formal developments and proof assistants have made it into the classroom. Paradoxically, in the automated reasoning community, we have not been very enthusiastic about formalizing our own results. This paper reported on some steps we have taken to address this.

Acknowledgement. We thank Tobias Nipkow for making this work possible. Jesper Bengtson, Nicholas Smallbone, Mark Summerfield, Dmitriy Traytel, and several anonymous reviewers suggested improvements to earlier versions of this paper. The research was supported by the Deutsche Forschungsgemeinschaft (DFG) projects Security Type Systems and Deduction (grant Ni 491/13-1), part of the program Reliably Secure Software Systems (RS^3, Priority Program 1496), and Hardening the Hammer (grant Ni 491/ 14-1). The authors are listed in alphabetical order.

References

[1] Ballarin, C.: Locales: A module system for mathematical theories. J. Autom. Reasoning (to appear)

[2] Berghofer, S.: First-order logic according to Fitting. In: Klein, G., Nipkow, T., Paulson, L. (eds.) Archive of Formal Proofs (2007), `http://afp.sf.net/entries/FOL-Fitting.shtml`

[3] Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 493–507. Springer, Heidelberg (2013)

[4] Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. Tech. report associated with TACAS 2013 paper [3] (2013), `http://www21.in.tum.de/~blanchet/enc_types_report.pdf`

[5] Blanchette, J.C., Krauss, A.: Monotonicity inference for higher-order formulas. J. Autom. Reasoning 47(4), 369–398 (2011)

[6] Blanchette, J.C., Popescu, A.: Formal development associated with this paper (2013), http://www21.in.tum.de/~popescua/fol_devel.zip

[7] Blanchette, J.C., Popescu, A.: Sound and complete sort encodings for first-order logic. In: Klein, G., Nipkow, T., Paulson, L. (eds.) Archive of Formal Proofs (2013), http://afp.sourceforge.net/entries/Sort_Encodings.shtml

[8] Blanchette, J.C., Popescu, A., Traytel, D.: Coinductive pearl: Modular first-order logic completeness (submitted), http://www21.in.tum.de/~blanchet/compl.pdf

[9] Bouillaguet, C., Kuncak, V., Wies, T., Zee, K., Rinard, M.: Using first-order theorem provers in the Jahob data structure verification system. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 74–88. Springer, Heidelberg (2007)

[10] Claessen, K., Lillieström, A., Smallbone, N.: Sort it out with monotonicity—Translating between many-sorted and unsorted first-order logic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 207–221. Springer, Heidelberg (2011)

[11] Goguen, J.A., Burstall, R.M.: Institutions: Abstract model theory for specification and programming. J. ACM 39(1), 95–146 (1992)

[12] Harrison, J.: Formalizing basic first order model theory. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 153–170. Springer, Heidelberg (1998)

[13] Harrison, J.: Towards self-verification of HOL Light. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 177–191. Springer, Heidelberg (2006)

[14] Huffman, B., Urban, C.: A new foundation for Nominal Isabelle. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 35–50. Springer, Heidelberg (2010)

[15] Kammüller, F., Wenzel, M.T., Paulson, L.C.: Locales - A sectioning concept for Isabelle. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 149–166. Springer, Heidelberg (1999)

[16] Monk, J.D.: Mathematical Logic. Springer (1976)

[17] Myreen, M.O., Davis, J.: A verified runtime for a verified theorem prover. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 265–280. Springer, Heidelberg (2011)

[18] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)

[19] 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)

[20] Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Wexelblat, R.L. (ed.) PLDI 1988, pp. 199–208. ACM (1988)

[21] Pitts, A.M.: Nominal logic, a first order theory of names and binding. Inf. Comput. 186(2), 165–193 (2003)

[22] Pitts, A.M.: Alpha-structural recursion and induction. J. ACM 53(3), 459–506 (2006)

[23] Popescu, A., Gunter, E.L.: Recursion principles for syntax with bindings and substitution. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) ICFP 2011, pp. 346–358. ACM (2011)

[24] Popescu, A., Gunter, E.L., Osborn, C.J.: Strong normalization of System F by HOAS on top of FOAS. In: LICS 2010, pp. 31–40. IEEE (2010)

[25] Ridge, T., Margetson, J.: A mechanically verified, sound and complete theorem prover for first order logic. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 294–309. Springer, Heidelberg (2005)

[26] Shankar, N.: Metamathematics, Machines, and Gödel's Proof. Cambridge Tracts in Theoretical Computer Science, vol. 38. Cambridge University Press (1994)

[27] Sutcliffe, G.: The 6th IJCAR automated theorem proving system competition—CASC-J6. AI Comm. 26(2), 211–223 (2013)

[28] Tinelli, C., Zarba, C.G.: Combining decision procedures for sorted theories. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 641–653. Springer, Heidelberg (2004)

From Resolution and DPLL to Solving Arithmetic Constraints

Konstantin Korovin*

School of Computer Science
The University of Manchester, UK
korovin@cs.man.ac.uk

Abstract. Reasoning methods based on resolution and DPLL have enjoyed many success stories in real-life applications. One of the challenges is whether we can go beyond and extend this technology to other domains such as arithmetic. In our recent work we introduced two methods for solving systems of linear inequalities called conflict resolution (CR) [6, 7] and bound propagation (BP) [3, 8] which aim to address this challenge. In particular, conflict resolution can be seen as a refinement of resolution and bound propagation is analogous to DPLL with constraint propagation, backjumping and lemma learning. There are non-trivial issues when considering arithmetic constraints such as termination, dynamic variable ordering and dealing with large coefficients. In this talk I will overview our approach and some related work [1, 2, 4, 5, 9]. This is a joint work with Ioan Dragan, Laura Kovács, Nestan Tsiskaridze and Andrei Voronkov.

References

1. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Mahboubi, A., Mebsout, A., Melquiond, G.: A simplex-based extension of Fourier-Motzkin for solving linear integer arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 67–81. Springer, Heidelberg (2012)
2. Cotton, S.: Natural Domain SMT: A Preliminary Assessment. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 77–91. Springer, Heidelberg (2010)
3. Dragan, I., Korovin, K., Kovács, L., Voronkov A.: Bound propagation for arithmetic reasoning in Vampire (submitted, 2013)
4. Jovanović, D., de Moura, L.: Cutting to the Chase Solving Linear Integer Arithmetic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 338–353. Springer, Heidelberg (2011)
5. 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)
6. Korovin, K., Tsiskaridze, N., Voronkov, A.: Conflict Resolution. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 509–523. Springer, Heidelberg (2009)
7. Korovin, K., Tsiskaridze, N., Voronkov, A.: Implementing conflict resolution. In: Clarke, E., Virbitskaite, I., Voronkov, A. (eds.) PSI 2011. LNCS, vol. 7162, pp. 362–376. Springer, Heidelberg (2012)

* Supported by a Royal Society University Fellowship.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 261–262, 2013.

8. Korovin, K., Voronkov, A.: Solving Systems of Linear Inequalities by Bound Propagation. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 369–383. Springer, Heidelberg (2011)
9. McMillan, K.L., Kuehlmann, A., Sagiv, M.: Generalizing DPLL to Richer Logics. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 462–476. Springer, Heidelberg (2009)

Tableaux for Relation-Changing Modal Logics

Carlos Areces[1,2], Raul Fervari[1], and Guillaume Hoffmann[1]

[1] FaMAF, Universidad Nacional de Córdoba, Argentina
{areces,fervari,hoffmann}@famaf.unc.edu.ar
[2] CONICET, Argentina

Abstract. We consider dynamic modal operators that can change the relation of a model during the evaluation of a formula. In this paper, we extend the basic modal language with modalities that are able to delete, add or swap pairs of related elements of the domain; and explore tableau calculi as satisfiability procedures for these logics.

1 Relation-Changing Modal Logics

We investigate modal operators that are suitable for reasoning about *dynamic aspects* of a given situation, e.g., how relations involving a set of elements *evolve* through time or through the application of certain operations. Instead of modeling the whole space of possible evolutions of the system as a graph, we use dynamic operators whose semantics directly correspond to the model evolutions that interest us. One example of such operators is *sabotage* introduced by Johan van Benthem in [8]. In the modal logic equipped with the sabotage operator, a formula can indicate that evaluation should continue in a model identical to the current one except that some edge has been removed from its relation.

In this article we present tableau methods for various relation-changing modal logics. We consider the basic modal logic $\mathcal{ML}$ [4] extended with the following operators: the local variant of sabotage $\langle sb\rangle$ deletes an arrow while traversing it; the *bridge* modality $\langle br\rangle$ adds an arrow from the current state of evaluation to a non-accessible state and continues the evaluation there; the *swap* modality $\langle sw\rangle$ inverts the direction of an arrow while traversing it. The swap modality was introduced in [3], and the local sabotage and bridge modalities in [2].

Definition 1 (Syntax). *Let* PROP *be a countable, infinite set of propositional symbols. The set* FORM *of formulas over* PROP *is defined as:*

$$\mathsf{FORM} ::= \bot \mid p \mid \neg\varphi \mid \varphi \wedge \psi \mid \blacklozenge\varphi,$$

where $p \in \mathsf{PROP}$*,* $\blacklozenge \in \{\Diamond, \langle sb\rangle, \langle br\rangle, \langle sw\rangle\}$ *and* $\varphi, \psi \in \mathsf{FORM}$*. Other operators are defined as usual. In particular,* $\blacksquare\varphi$ *is defined as* $\neg\blacklozenge\neg\varphi$*.*

Formulas of the basic modal language $\mathcal{ML}$ *contains only* $\Diamond$ *besides the Boolean operators. We call* $\mathcal{ML}(\blacklozenge)$ *the extension of* $\mathcal{ML}$ *allowing also the* $\blacklozenge$ *operator, for* $\blacklozenge \in \{\langle sb\rangle, \langle br\rangle, \langle sw\rangle\}$*.*

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 263–278, 2013.

Semantically, formulas are evaluated in standard relational models, and the meaning of the basic modal operators is unchanged. When we evaluate formulas containing dynamic operators, we need to keep track of the edges that have been modified. To that end, let us define precisely the models that we use.

Definition 2 (Models and Model Variants). *A model is a triple $\mathcal{M} = \langle W, R, V\rangle$, where W is a non-empty set whose elements are called states; $R \subseteq W \times W$ is the accessibility relation; and $V : \mathsf{PROP} \mapsto \mathcal{P}(W)$ is a valuation.*

Given a model $\mathcal{M} = \langle W, R, V\rangle$ we define the following notation:

(sabotaging) $\mathcal{M}^-_S = \langle W, R^-_S, V\rangle$, *with* $R^-_S = R \backslash S$, $S \subseteq R$.
(bridging) $\mathcal{M}^+_S = \langle W, R^+_S, V\rangle$, *with* $R^+_S = R \cup S$, $S \subseteq (W \times W)\backslash R$.
(swapping) $\mathcal{M}^*_S = \langle W, R^*_S, V\rangle$, *with* $R^*_S = (R\backslash S^{-1}) \cup S$, $S \subseteq W \times W$.

Let w be a state in $\mathcal{M}$, the pair $(\mathcal{M}, w)$ is called a pointed model*; we will usually drop parenthesis and call $\mathcal{M}, w$ a pointed model. A* model variant *of $\mathcal{M}$ is a model obtained from $\mathcal{M}$ by some of the above operations.*

In the rest of this article we will use wv as a shorthand for $\{(w, v)\}$ or (w, v).

Definition 3 (Semantics). *Given a pointed model $\mathcal{M}, w$ and a formula φ we say that $\mathcal{M}, w$ satisfies φ, and write $\mathcal{M}, w \models \varphi$, when*

$$\begin{array}{lll}
\mathcal{M}, w \models p & \textit{iff} & w \in V(p) \\
\mathcal{M}, w \models \neg\varphi & \textit{iff} & \mathcal{M}, w \not\models \varphi \\
\mathcal{M}, w \models \varphi \wedge \psi & \textit{iff} & \mathcal{M}, w \models \varphi \textit{ and } \mathcal{M}, w \models \psi \\
\mathcal{M}, w \models \Diamond\varphi & \textit{iff} & \textit{for some } v \in W \textit{ s.t. } Rwv, \quad \mathcal{M}, v \models \varphi \\
\mathcal{M}, w \models \langle sb\rangle\varphi & \textit{iff} & \textit{for some } v \in W \textit{ s.t. } Rwv, \quad \mathcal{M}^-_{wv}, v \models \varphi \\
\mathcal{M}, w \models \langle br\rangle\varphi & \textit{iff} & \textit{for some } v \in W \textit{ s.t. } \neg Rwv, \mathcal{M}^+_{wv}, v \models \varphi \\
\mathcal{M}, w \models \langle sw\rangle\varphi & \textit{iff} & \textit{for some } v \in W \textit{ s.t. } Rwv, \quad \mathcal{M}^*_{vw}, v \models \varphi
\end{array}$$

φ is satisfiable *if for some pointed model $\mathcal{M}, w$ we have $\mathcal{M}, w \models \varphi$.*

Adding any of the previous operators to the basic modal logic increases its expressive power. A basic result for $\mathcal{ML}$ [4] shows that it has the *tree model property*: every satisfiable formula of $\mathcal{ML}$ can be satisfied at the root of a model where the accessibility relation defines a tree. In [2] we introduced formulas using the operators above that cannot be satisfied at the root of a tree:

1. $\varphi = \Diamond\Diamond\top \;\wedge\; [sb]\Box\bot$ is true at a state w, only if w is reflexive.
   Suppose we evaluate φ at some state w of an arbitrary model. On one hand, the 'static' part of the formula $\Diamond\Diamond\top$ ensures it is possible to take two steps using the accessibility relation. On the other hand, the 'dynamic' part of the formula $[sb]\Box\bot$ tells us that after traversing any edge and eliminating it we arrive to a dead-end. This can only happen if the state w is reflexive and does not have any other outgoing links.
2. $\varphi = \Box\bot \wedge \langle br\rangle\langle br\rangle\top$ is only satisfiable in models where the root is a dead-end and there is a second, unreachable state.

3. $\varphi = p \wedge (\bigwedge_{1\leq i\leq 3} \Box^i \neg p) \wedge \langle sw\rangle \Diamond\Diamond p$ is true at a state w, only if w has a reflexive successor.

   Suppose we evaluate φ at a state w in a model. The 'static' part of the formula $p \wedge (\bigwedge_{1\leq i\leq 3} \Box^i \neg p)$ makes p true in w and ensures that no p state is reachable within three steps from w (also w cannot be reflexive). Because $\langle sw\rangle \Diamond\Diamond p$ is true at w, there is an R-successor v where $\Diamond\Diamond p$ holds once the accessibility relation has been updated to R^*_{vw}. That is, v has to reach a p-state in exactly two R^*_{vw}-steps. The only p-state sufficiently close is w which is reachable in one step. As w is not reflexive, v has to be reflexive so that we can linger at v for one loop and reach p in the correct number of steps.

With respect to computational complexity, satisfiability of $\mathcal{ML}(\langle sw\rangle)$ is known to be undecidable [3], and we conjecture that the same holds for the other two logics. The finite model property fails for the three logics. For this reason, and as we will not introduce control mechanism like loop checks, the tableau procedures we will define not necessarily terminate on all inputs.

In Section 2 we will introduce complete and sound tableau calculi for these logics. In Section 3 we extend the results to the global counterparts of the operators. In Section 4 we discuss a few final issues.

2 Tableau Calculi

We present basic definitions for different tableau algorithms for the relation-changing modal logics we introduced in the previous section. These algorithms will rely on the same data structures and will only differ in some of their rules.

Definition 4 (Tableau formulas). *Let* NOM *be an infinite, well ordered set of symbols we call* nominals*. A* tableau formula *is either a* prefixed formula*, an* equational formula *or a* relational formula*. A prefixed formula is of the form* $(n, X) : \varphi$*, with* $n \in \mathsf{NOM}$*,* $X \subseteq \mathsf{NOM}^2$*, and* φ *a formula of the considered object language. An equational formula is a Boolean combination of formulas of the form* $n \doteq m$ *or* $n \dot{\neq} m$ *for* $n, m \in \mathsf{NOM}$*. We also use the following notation:*

$$\begin{array}{rclcrcl} nm \doteq xy & := & n \doteq x \;\wedge\; m \doteq y & \quad & nm \dot{\in} X & := & \bigvee_{xy \in X} nm \doteq xy \\ nm \dot{\neq} xy & := & n \dot{\neq} x \;\vee\; m \dot{\neq} y & \quad & nm \dot{\notin} X & := & \bigwedge_{xy \in X} nm \dot{\neq} xy. \end{array}$$

In particular $nm \dot{\in} \emptyset$ *is a notation for* $\bot$ *and* $nm \dot{\notin} \emptyset$ *is a notation for* $\top$*. A relational formula is of the form* $\dot{R}nm$ *or* $\neg \dot{R}nm$*, with* $n, m \in \mathsf{NOM}$*.*

The set X of a prefixed formula $(n, X) : \varphi$ is used to describe the model variant in which the formula φ is to be interpreted. According to the logic we are in this set is to be interpreted differently. This is done by fixing a function f that, out of a relation $R, S \subseteq W \times W$ yields another relation $R' = f(R, S)$.

Definition 5 (Branches and interpretations). *A* branch *is a non-empty set of tableau formulas. Let $\mathcal{M} = \langle W, R, V\rangle$ be a model, $f : W^2 \times W^2 \mapsto W^2$ a relation-changing function and $\sigma : \mathsf{NOM} \mapsto W$ a mapping from nominals to states of $\mathcal{M}$. Let $X^\sigma = \{\sigma(a)\sigma(b) \mid ab \in X\}$, for $X \subseteq \mathsf{NOM}^2$.*

Given $\mathcal{M} = \langle W, R, V\rangle$, let $\mathcal{M}^f_{X^\sigma} = \langle W, f(R, X^\sigma), V\rangle$. That is, $\mathcal{M}^f_{X^\sigma}$ is the model $\mathcal{M}$ updated by the relation-changing function f according to a set of pairs of nominals X under mapping σ.

A branch Θ is satisfiable *if there exists a model $\mathcal{M} = \langle W, R, V\rangle$ and a mapping σ such that all the formulas of Θ are satisfiable under model $\mathcal{M}$ and mapping σ. That is, they should satisfy the following conditions:*

- *if $(n, X) : \varphi \in \Theta$ then $\mathcal{M}^f_{X^\sigma}, \sigma(n) \models \varphi$,*
- *if $n \dot{=} m \in \Theta$ then $\sigma(n) = \sigma(m)$,*
- *if $n \dot{\neq} m \in \Theta$ then $\sigma(n) \neq \sigma(m)$,*
- *Boolean combinations of equational formulas are interpreted as expected,*
- *if $\dot{R}nm \in \Theta$ then $R\sigma(n)\sigma(m)$,*
- *if $\neg\dot{R}nm \in \Theta$ then $\neg R\sigma(n)\sigma(m)$.*

A branch is unsatisfiable *if it is not satisfiable.*

A *tableau calculus* is a set of rules such that each rule applies to a branch and yields one or more branches, under certain conditions. These conditions are called saturation conditions, and stipulate that no rule can be applied twice on the same premises, and that no formula can be introduced twice in a branch.

A *tableau* is a tree in which each node defines a tableau branch, and edges represent applications of tableau rules. A tableau is expanded as much as possible by the rules of the system (i.e., rules are applied whenever possible according to the saturation condition). A fully expanded branch is called *saturated.*

A tableau branch is *closed* if it contains $\bot$, otherwise it is *open.* A tableau is closed if all branches are closed, otherwise it is open.

Given a branch Θ, $\sim_\Theta$ denotes the equivalence closure of the relation $\{nm \mid n \dot{=} m \in \Theta\}$, and we write $\bar{n}$ for the smallest nominal x such that $x \sim_\Theta n$. For $X \subseteq \mathsf{NOM}^2$ we write $\bar{X} = \{\bar{n}\bar{m} \mid nm \in X\}$. Figure 1 presents the rules common to all the tableau calculus of this work. They are the Boolean rules $(\wedge)$ and $(\vee)$, the clashing rules $(\bot_{atom})$ and $(\bot_{\neq})$, the equational rules $(R\sim)$ and (Id), and the unrestricted blocking rule (ub) [7]. We use the unrestricted blocking rule as a way to saturate branches with equational formulas. These formulas can appear as premises of tableau rules in the calculi we introduce later.

This result follows easily from the tableau rules:

Lemma 6. *Let Θ be a saturated open branch. If $nm \dot{\in} S$ is in Θ then $\bar{n}\bar{m} \in \bar{S}$. If $nm \dot{\notin} S$ is in Θ then $\bar{n}\bar{m} \notin \bar{S}$.*

When it comes to adequacy of a tableau calculus, we have to consider two properties: completeness and soundness. Given a tableau calculus $\mathcal{T}$, let us write $\mathcal{T}(\varphi)$ to refer to a tableau obtained by running $\mathcal{T}$ on the input formula $(n_0, \emptyset) : \varphi$, where n_0 is the smallest nominal in NOM. Then we define:

$$\frac{(n,X):\ \varphi\wedge\psi}{\begin{array}{c}(n,X):\varphi\\(n,X):\psi\end{array}}\,(\wedge) \qquad \frac{(n,X):\ \varphi\vee\psi}{(n,X):\varphi \mid (n,X):\psi}\,(\vee)$$

$$\frac{\begin{array}{c}(n,X_1):p\\(n,X_2):\neg p\end{array}}{\bot}\,(\bot_{atom})^1 \qquad \frac{\begin{array}{c}n\sim_\Theta m\\n\dot{\neq}m\end{array}}{\bot}\,(\bot_{\neq})$$

$$\frac{\dot{R}nm}{\dot{R}\bar{n}\bar{m}}\,(R\sim) \qquad \frac{(n,X):\varphi}{(\bar{n},X):\varphi}\,(Id) \qquad \frac{}{n\doteq m \mid n\dot{\neq}m}\,(ub)^2$$

[1] $p \in \mathsf{PROP}$

[2] n and m are two different nominals in the branch

Fig. 1. Common tableau rules

Definition 7 (Completeness). *A tableau calculus $\mathcal{T}$ is* complete *if for any formula φ, if $\mathcal{T}(\varphi)$ is open then φ is satisfiable.*

Definition 8 (Soundness). *A tableau calculus $\mathcal{T}$ is* sound *if for any formula φ, if φ is satisfiable then $\mathcal{T}(\varphi)$ is open.*

We define models induced from open branches.

Definition 9 (Induced Models). *Let Θ be an open branch. We define $\mathcal{M}^\Theta = \langle W^\Theta, R^\Theta, V^\Theta\rangle$, the* induced model for Θ*, as:*

$$\begin{array}{ll} W^\Theta &= \{\bar{n} \mid n \in \Theta\}\\ R^\Theta &= \{(\bar{n},\bar{m}) \mid \dot{R}nm \in \Theta\}\\ V^\Theta(p) &= \{\bar{n} \mid n : p \in \Theta\}. \end{array}$$

We want to show that a tableau system is sound and complete, i.e., that for any formula φ, $\mathcal{T}(\varphi)$ is open if, and only if, φ is satisfiable. Moreover, if $\mathcal{T}(\varphi)$ has an open branch Θ then $\mathcal{M}^\Theta$ is a model that satisfies φ. We present tableau calculi for $\mathcal{ML}(\langle sb\rangle)$, $\mathcal{ML}(\langle br\rangle)$ and $\mathcal{ML}(\langle sw\rangle)$ in the next sections.

2.1 Sabotage

Figure 2 introduces rules that, in combination with those in Figure 1, form a complete and sound tableau calculus for $\mathcal{ML}(\langle sb\rangle)$. In this calculus, a formula $(n,S):\varphi$ is understood as "φ holds at the state referred to by n in the model variant described by the set of sabotaged pairs S".

We interpret branches of this tableau calculus with the following relation-changing function: $f : (R,S) \mapsto R \setminus S$. This means that a formula $(n,S):\varphi$ in a branch Θ should hold in the induced model variant $\mathcal{M}^\Theta_S$ defined as $\mathcal{M}^\Theta_S = \langle W^\Theta, R^\Theta_S, V^\Theta\rangle$, where $R^\Theta_S = R^\Theta \ \setminus\ \bar{S}$.

$$\frac{(n,S):\Diamond\varphi}{\begin{array}{c}\dot{R}nm\\ nm\dot{\notin}S\\ (m,S):\varphi\end{array}}\ (\Diamond)^1 \qquad\qquad \frac{\begin{array}{c}(n,S):\Box\varphi\\ \dot{R}nm\\ nm\dot{\notin}S\end{array}}{(m,S):\varphi}\ (\Box)$$

$$\frac{(n,S):\langle sb\rangle\varphi}{\begin{array}{c}\dot{R}nm\\ nm\dot{\notin}S\\ (m,S\cup nm):\varphi\end{array}}\ (\langle sb\rangle)^1 \qquad\qquad \frac{\begin{array}{c}(n,S):[sb]\varphi\\ \dot{R}nm\\ nm\dot{\notin}S\end{array}}{(m,S\cup nm):\varphi}\ ([sb])$$

[1] m is new.

Fig. 2. Tableau rules for $\mathcal{ML}(\langle sb\rangle)$

The rules involve the notation $nm\dot{\notin}S$. $nm\dot{\notin}S$ specifies that the edge referred to by the pair of nominals (n,m) should not be deleted in the model variant described by S. When present as premise of a rule, this condition requires that one of the disjuncts in $nm\dot{\notin}S$ is present in the branch, which in turn means that either $n\dot{\neq}x$ or $m\dot{\neq}y$ is in the branch for all $xy \in S$.

The $(\Diamond)$ rule captures the standard meaning of the $\Diamond$ connector, but adds a new constraint that specifies that the successor has not been deleted at this point of the branch. $(\Box)$ should also take this into account. For each successor m of n in the initial model $(\dot{R}nm)$, and only if it the edge between n and m has not been sabotaged $(nm\dot{\notin}S)$, φ must hold at m in the same model variant. Rule $([sb])$ is similar to $(\Box)$, but φ must hold at m in the model variant where the edge nm is sabotaged. Rule $(\langle sb\rangle)$ corresponds similarly to $(\Diamond)$.

Figure 3 presents an example with a satisfiable formula of $\mathcal{ML}(\langle sb\rangle)$.

We will now prove completeness and soundness of the calculus for $\mathcal{ML}(\langle sb\rangle)$.

Lemma 10. *Let Θ be a saturated, open branch and φ an $\mathcal{ML}(\langle sb\rangle)$-formula. If $(n,S):\varphi \in \Theta$ then $\mathcal{M}^{\Theta}_{S},\bar{n} \models \varphi$.*

Proof. Let $(n,S):\varphi \in \Theta$. Proceed by structural induction on φ.

- $\boldsymbol{p}$**:** By definition, $\bar{n} \in V^{\Theta}(p)$, then $\mathcal{M}^{\Theta},\bar{n} \models p$ and $\mathcal{M}^{\Theta}_{S},\bar{n} \models p$.
- $\boldsymbol{\neg p}$**:** By saturation of (Id), $\bar{n} : \neg p \in \Theta$. Since Θ is open, $\bar{n} : p \notin \Theta$. By definition, $\bar{n} \notin V^{\Theta}(p)$, then $\mathcal{M}^{\Theta},\bar{n} \not\models p$ and $\mathcal{M}^{\Theta}_{S},\bar{n} \not\models p$.
- $\boldsymbol{\psi \wedge \chi}$ **and** $\boldsymbol{\psi \vee \chi}$: Trivial by inductive hypothesis.
- $\boldsymbol{\Diamond\psi}$**:** By $(\Diamond)$, Θ contains $\dot{R}nm$, $nm\dot{\notin}S$ and $(m,S):\psi$. We want to show that $\bar{n}\bar{m} \in R^{\Theta}_{S}$. We verify the following:
  1. $\bar{n}\bar{m} \in R^{\Theta}$: this is true since $\dot{R}nm \in \Theta$.
  2. $\bar{n}\bar{m} \notin \bar{S}$: this is true since $(nm\dot{\notin}S) \in \Theta$ by Lemma 6.

  Since $\bar{n}\bar{m} \in R^{\Theta}_{S}$, and (by (Id)) $(\bar{m},S):\psi \in \Theta$, we have $\mathcal{M}^{\Theta}_{S},\bar{n} \models \Diamond\psi$.
- $\boldsymbol{\langle sb\rangle\psi}$**:** We need to show that $\mathcal{M}^{\Theta}_{S},\bar{n} \models \langle sb\rangle\psi$, i.e., there exists $x \in V^{\Theta}$ s.t $\mathcal{M}^{\Theta}_{S\cup pq},x \models \psi$, where $\bar{p} = \bar{n}$ and $\bar{q} = x$. This can be checked considering $(\langle sb\rangle)$ instead of $(\Diamond)$ as for the previous case.

Example: A tableau for $\Diamond\Diamond\top \ \wedge \ [sb]\Box\bot$ follows:

(1)	$(n_0, \emptyset) : \Diamond\Diamond\top \ \wedge \ [sb]\Box\bot$		initial node
(2)	$(n_0, \emptyset) : \Diamond\Diamond\top$		$(\wedge)$ on (1)
(3)	$(n_0, \emptyset) : [sb]\Box\bot$		
(4)	$\dot{R}n_0n_1$		$(\Diamond)$ on (2)
(5)	$n_0n_1\dot{\notin}\emptyset$		
(6)	$(n_1, \emptyset) : \Diamond\top$		
(7)	$\dot{R}n_1n_2$		$(\Diamond)$ on (6)
(8)	$n_1n_2\dot{\notin}\emptyset$		
(9)	$(n_2, \emptyset) : \top$		
(10)	$(n_1, \{n_0n_1\}) : \Box\bot$		$([sb])$ on (3) and (4) with trivial condition $n_0n_1\dot{\notin}\emptyset$
(11)	$n_0\dot{=}n_1$	$n_0\dot{\neq}n_1$	(ub)

The right branch soon closes since $(\Box)$ applies on (10) and (7) with condition $n_1n_2\dot{\notin}\{n_0, n_1\}$ fulfilled by $n_0\dot{\neq}n_1$, and introduces $\bot$. Let us expand the left branch:

(12)	$n_1\dot{=}n_2$	$n_1\dot{\neq}n_2$	(ub)

Again the right branch closes by application of $(\Box)$ with condition $n_1n_2\dot{\notin}\{n_0, n_1\}$ fulfilled by $n_1\dot{\neq}n_2$. We expand the left branch:

(13)	$n_0\dot{=}n_2$	$n_0\dot{\neq}n_2$	(ub)

The right branch above closes by rule $(\bot_{\neq})$. Left branch is saturated and open, with the following induced model:

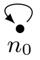


Fig. 3. Tableau example for $\mathcal{ML}(\langle sb \rangle)$

- $\Box\psi$**:** We only consider states $x \in W^{\Theta}$ such that $\bar{n}x \in R^{\Theta}_S$. That is, there exists a, b such that $\dot{R}ab \in \Theta$ and $\bar{n}x = \bar{a}\bar{b}$, and $\bar{n}x \notin \bar{S}$. The condition of rule $(\Box)$ $(nm\dot{\notin}S)$ does not prevent it from being applied on such pair of nominals. By (Id), $(\bar{n}, S) : \Box\psi \in \Theta$, i.e., $(\bar{a}, S) : \Box\psi \in \Theta$, and also by $(R{\sim})$, $\dot{R}\bar{a}\bar{b} \in \Theta$. By $(\Box)$ we have $(\bar{b}, S) : \psi \in \Theta$. Now, $\bar{b} = x$, so $\mathcal{M}^{\Theta}_S, x \models \psi$. Hence for all $x \in V^{\Theta}$ such that $\bar{n}x \in R^{\Theta}_S$, $\mathcal{M}^{\Theta}_S, x \models \psi$, i.e., $\mathcal{M}^{\Theta}_S, \bar{n} \models \Box\psi$.
- $[sb]\psi$**:** We need to show that $\mathcal{M}^{\Theta}_S, \bar{n} \models [sb]\psi$, i.e., for all $x \in W^{\Theta}$ such that $(\bar{n}, x) \in R^{\Theta}_S$, $\mathcal{M}^{\Theta}_{S \cup pq}, x \models \psi$, where $\overline{pq} = \bar{n}x$. This can be checked considering rule $([sb])$ instead of $(\Box)$ as for the previous case. □

By the previous lemma we get:

Theorem 11 (Completeness). *If $\mathcal{T}(\varphi)$ is open, then φ is satisfiable.*

We now show soundness of the calculus for $\mathcal{ML}(\langle sb \rangle)$.

Lemma 12. *Let Γ be a set of satisfiable tableau formulas, and $\varphi \in \mathcal{ML}(\langle sb \rangle)$. If there is a closed tableau $\mathcal{T}(\Gamma')$ for $\Gamma' = (\Gamma \cup \{\neg\varphi\})$, then φ is satisfiable.*

Proof. Let $\varTheta$ be a satisfiable branch. Following Definition 5, $\varTheta$ is satisfied by a model $\mathcal{M} = \langle W, R, V\rangle$ and a mapping $\sigma : \mathsf{NOM} \mapsto W$. We write $\sigma[m \mapsto v]$ to refer to the mapping equal to σ except, perhaps, $\sigma(m) = v$.

Assume that there is a closed tableau $\mathcal{T}(\varGamma')$ such that $\varGamma' = (\varGamma \cup \{\neg\varphi\})$. We will prove $\varGamma'$ unsatisfiable, by induction on the tableau structure.

- **($\bot_{\boldsymbol{atom}}$):** If this rule applies, then $n : a \in \varGamma'$ and $n : \neg a \in \varGamma'$, for some n, a. Then $\varGamma'$ is trivially unsatisfiable.
- Common rules $(\bot_{\neq})$, $(\wedge)$, $(\vee)$, $(R\sim)$, (Id) and (ub) are easy to check.

It remains to verify, for each remaining rule, that their application to a satisfiable branch generates at least one satisfiable branch. In the present calculus, all remaining rules are non-branching.

- **($\Diamond$):** Suppose $(n, S) : \Diamond\varphi \in \mathcal{T}(\varGamma')$. We know that $(n, S) : \Diamond\varphi$ is satisfiable, then there is a model $\mathcal{M} = \langle W, R, V\rangle$, and a mapping $\sigma : \mathsf{NOM} \mapsto W'$ s.t. $\mathcal{M}^-_{S^\sigma}, \sigma(n) \models \Diamond\varphi$. By definition of $\models$, there exists $v \in W$ s.t. $\sigma(n)v \in R \setminus S^\sigma$ and $\mathcal{M}^-_{S^\sigma}, v \models \varphi$. The $(\Diamond)$ rule generates $\dot{R}nm$, $nm \dot{\notin} S$ and $(m, S) : \varphi$, with m new in the branch. We need to check that the branch containing these three new formulas is satisfiable. That is, there exists a model and a mapping satisfying them. Let us consider the mapping $\sigma' = \sigma[m \mapsto v]$ and check that the interpretation $\mathcal{M}, \sigma'$ satisfies the new branch:
  - $\dot{R}nm$ is satisfied since $R\sigma'(n)\sigma'(m)$, i.e., $R\sigma(n)v$, holds.
  - Consider $nm \dot{\notin} S$. It suffices to check that for all $xy \in S$, $\sigma'(n)\sigma'(m) \neq \sigma'(x)\sigma'(y)$, i.e., $\sigma(n)v \neq \sigma'(x)\sigma'(y)$. But $\sigma(n)v = \sigma'(x)\sigma'(y)$ would contradict $\sigma(n)v \in R \setminus S^\sigma$.
  - $\mathcal{M}, \sigma'$ satisfies $(m, S) : \varphi$ since $\mathcal{M}^-_{S^{\sigma'}}, \sigma'(m) \models \varphi$ holds.
- **($\langle \boldsymbol{sb} \rangle$):** This case is similar to $(\Diamond)$, except that we need to check that the new tableau formula $(m, S \cup nm) : \varphi$ is satisfied. This is done considering the new mapping $\sigma' = \sigma[m \mapsto v]$ and observing that $\mathcal{M}^-_{S \cup nm^{\sigma'}}, \sigma'(m) \models \varphi$.
- **($\Box$):** Suppose $(n, S) : \Box\varphi$ and $\dot{R}nm$ are in $\varTheta$, and the condition $nm \dot{\notin} S$ holds. This implies that there exists $\mathcal{M} = \langle W, R, V\rangle$ and a mapping σ such that $\mathcal{M}^-_{S^\sigma}, \sigma(n) \models \Box\varphi$, and $R\sigma(n)\sigma(m)$, and there is no pair of nominals $xy \in S$ such that $nm = xy$. This means that for all $v \in W$ s.t. $\sigma(n)v \in (R \setminus S^\sigma)$, $\mathcal{M}^-_{S^\sigma}, v \models \varphi$ and there exists $v \in W$ s.t. $R\sigma(n)v$. We verify that $(m, S) : \varphi$ is satisfied by $\mathcal{M}, \sigma$. Since $\sigma(n)\sigma(m) \in (R \setminus S^\sigma)$, then $\mathcal{M}^-_{S^\sigma}, \sigma(m) \models \varphi$. Hence $(m, S) : \varphi$ is satisfied by $\mathcal{M}, \sigma$.
- **($[\boldsymbol{sb}]$):** This is similar to the $(\Box)$ case, but we have to show that $(m, S \cup nm) : \varphi$ is satisfied by $\mathcal{M}, \sigma$. This is done by observing that if $\mathcal{M}^-_{S^\sigma}, \sigma(n) \models [sb]\varphi$ and $R\sigma(n)\sigma(m)$ then $\mathcal{M}^-_{S \cup nm^\sigma}, \sigma(m) \models \varphi$. □

From the previous lemma we get the following result:

Theorem 13 (Soundness). *If φ is satisfiable, then $\mathcal{T}(\varphi)$ is open.*

$$\frac{(n,B):\Diamond\varphi}{\begin{array}{c|c}\dot{R}nm & nm\dot{\in}B\\ (m,B):\varphi & (m,B):\varphi\end{array}}\,(\Diamond)^1 \qquad \frac{\begin{array}{c}(n,B):\Box\varphi\\ \dot{R}nm\end{array}}{(m,B):\varphi}\,(\Box) \qquad \frac{\begin{array}{c}(n,B):\Box\varphi\\ nm\dot{\in}B\end{array}}{(m,B):\varphi}\,(\Box_2)$$

$$\frac{\begin{array}{c}\dot{R}nm\\ (a,B):\varphi\\ nm\dot{\in}B\end{array}}{\bot}\,(R_\bot) \qquad \frac{(n,B):\langle br\rangle\varphi}{\begin{array}{c}nm\dot{\notin}B\\ (m,B\cup nm):\varphi\end{array}}\,(\langle br\rangle)^1 \qquad \frac{\begin{array}{c}(n,B):[br]\varphi\\ nm\dot{\notin}B\end{array}}{(m,B\cup nm):\varphi \mid \dot{R}nm}\,([br])^2$$

[1] m is new.
[2] m is already in the branch.

Fig. 4. Tableau rules for $\mathcal{ML}(\langle br\rangle)$

2.2 Bridge

Figure 4 presents rules for the tableau calculus corresponding to $\mathcal{ML}(\langle br\rangle)$ which should be combined with the common rules of Figure 1. The main difference with rules for sabotage is that they use as prefix a set of pairs of nominals B to keep track of edges that have been added to the relation of the original model.

The interpretation function will be $f : (R, B) \mapsto R \cup B$. This means that a formula $(n, B) : \varphi$ in a branch Θ should hold in the induced model variant $\mathcal{M}^{\Theta}_{B}$ defined as $\mathcal{M}^{\Theta}_{B} = \langle W^{\Theta}, R^{\Theta}_{B}, V^{\Theta}\rangle$, where $R^{\Theta}_{B} = R^{\Theta} \ \cup \ \bar{B}$. The notation $nm\dot{\in}B$ means that the edge represented by the nominals n and m is one of the edges added since the initial model in the model variant described by B. When used as a premise of a rule, the condition $nm\dot{\in}B$ requires that there exists some $xy \in B$ such that $n\dot{=}x$ and $m\dot{=}y$ are present in the branch. $nm\dot{\notin}B$ means that the edge (n, m) has not been added since the initial model in the variant described by B.

Some rules are more involved in this calculus. The rule $(\Diamond)$, when applied on a formula $(n, B) : \Diamond\varphi$, has to ensure that in the model variant described by B, the state referred to by the nominal n has a successor where φ holds. This model variant has a relation that is the union of the relation in the initial model and B. This is why $(\Diamond)$ is a branching rule that either chooses that the edge (n, m) belongs to the initial relation or to B.

The rule $(\Box)$ is the standard box rule for the basic modal logic. It is completed by a $(\Box_2)$ rule that ensures new edges of model variants are taken into account.

The new clash rule $(R_\bot)$ ensures that whenever some edge nm is present in a set of new edges B representing some model variant, the same edge is not present in the original model, i.e., $\dot{R}nm$ is forbidden to occur in the branch.

The rule $(\langle br\rangle)$ differs from $(\Diamond)$. This is because the $\langle br\rangle$ operator jumps to a state that should *not* be accessible from the current state, hence the introduction of $nm\dot{\notin}B$ and $(m, B \cup nm) : \varphi$ to the branch. This last formula, together with rule $(R_\bot)$, ensures that the edge nm is not in the original model.

The rule $([br])$ branches when applied to a formula $(n, B) : [br]\varphi$. It decides, for every nominal m such that $nm\dot{\notin}B$, whether $(m, R \cup nm) : \varphi$ holds, or $\dot{R}nm$

holds. In the first case, together with rule $(R_\perp)$, it ensures that the edge nm is not in the original model. In the second case, it ensures the contrary, hence no bridging to m is possible and φ does not need to hold at m.

Completeness and soundness of this tableau calculus can be proved as in the previous section. Figure 5 shows an example of how the rules are used.

Example: Consider the satisfiable formula $p \wedge \Diamond\neg p \wedge [br]p$. In the following tableau we hide the branches that directly close by vacuity of quantification:

(1)	$(n_0, \emptyset) : p \ \wedge \ \Diamond\neg p \ \wedge \ [br]p$		initial node
(2)	$(n_0, \emptyset) : p$		$(\wedge)$ on (1)
(3)	$(n_0, \emptyset) : \Diamond\neg p$		
(4)	$(n_0, \emptyset) : [br]p$		
(5)	$\dot{R}n_0n_1, (n_1, \emptyset) : \neg p$		$(\Diamond)$ on (3)
(6)	$n_0 \dot{=} n_1$	$n_0 \dot{\neq} n_1$	(ub)

Left branch closes due to (Id) and $(\perp_{atom})$. Right branch:

(7)	$(n_1, \{n_0n_1\}) : p$	$\dot{R}n_0n_1$	$([br])$ on (3), n_1

Left branch closes by $(\perp_{atom})$ on $(n_1, \{n_0n_1\}) : p$ and $(n_1, \emptyset) : \neg p$. Right branch:

(8)	$(n_0, \{n_0n_0\}) : p$	$\dot{R}n_0n_0$	$([br])$ on (4), n_0

Both branches are open and saturated. We have the following two induced models:

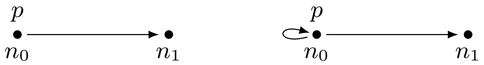


Fig. 5. Tableau example for $\mathcal{ML}(\langle br \rangle)$

2.3 Swap

Rules for the swap calculus are given in Figure 6, to be used in combination with the rules in Figure 1.

These rules have to handle the fact that swapping edges in a model can make some edges of the original model no longer usable (as when using the sabotage modality), and can make new edges usable (as with bridge). The set S that prefixes formulas of the calculus has to be understood as the pairs of states that no longer are part of the relation of the model variant. S^{-1} contains the edges that should be added to the model.

The interpretation function for this calculus is $f : (R, S) \mapsto (R \backslash S) \cup S^{-1}$. This means that a formula $(n, S) : \varphi$ in a branch Θ should hold in the induced model variant $\mathcal{M}^\Theta_S$ defined as $\mathcal{M}^\Theta_S = \langle W^\Theta, R^\Theta_S, V^\Theta \rangle$, where $R^\Theta_S = (R^\Theta \ \backslash \ \bar{S}) \ \cup \ \bar{S}^{-1}$.

In this calculus, S is kept irreflexive and asymmetric. Moreover, it will not contain two different pairs of nominals that refer to the same edge in the induced model. This guarantees that the names in S can be manipulated by the calculus as expected, in particular when a swapped edge must be swapped again. $nm \dot{\in} S$

$$\frac{(n,S):\Diamond\varphi}{\begin{array}{c|c} \dot{R}nm & nm\dot{\in}S^{-1} \\ nm\dot{\notin}S & (m,S):\varphi \\ (m,S):\varphi & \end{array}}\,(\Diamond)^{1} \qquad \frac{\begin{array}{c}(n,S):\Box\varphi \\ \dot{R}nm \\ nm\dot{\notin}S\end{array}}{(m,S):\varphi}\,(\Box) \qquad \frac{\begin{array}{c}(n,S):\Box\varphi \\ nm\dot{\in}S^{-1}\end{array}}{(m,S):\varphi}\,(\Box_2)$$

$$\frac{\begin{array}{c}(n,S):[sw]\varphi \\ \dot{R}nn\end{array}}{(n,S):\varphi}\,([sw]) \qquad \frac{\begin{array}{c}(n,S):[sw]\varphi \\ \dot{R}nm \\ n\dot{\neq}m \\ nm\dot{\notin}(S\cup S^{-1})\end{array}}{(m,S\cup nm):\varphi}\,([sw]_2) \qquad \frac{\begin{array}{c}(n,S):[sw]\varphi \\ xy\in S \\ n\dot{=}y\end{array}}{(x,S\backslash xy\cup yx):\varphi}\,([sw]_3)$$

$$\frac{(n,S):\langle sw\rangle\varphi}{\begin{array}{c|c|c} \dot{R}nn & \dot{R}nm & \bigvee_{xy\in S}(n\dot{=}y \wedge (x,S\backslash xy\cup yx){:}\varphi) \\ (n,S):\varphi & n\dot{\neq}m & \\ & nm\dot{\notin}(S\cup S^{-1}) & \\ & (m,S\cup nm):\varphi & \end{array}}\,(\langle sw\rangle)^{1}$$

[1] m is new.

Fig. 6. Tableau rules for $\mathcal{ML}(\langle sw\rangle)$

means that nm is no longer present in the model variant represented by S. $nm\dot{\in}S^{-1}$ means that nm has been added to the S model variant.

Let us examine the rules. $(\Diamond)$ is a combination of the $(\Diamond)$ rules for sabotage and bridge. It satisfies the formula $(n,S):\varphi$ in a state that is either accessible through the initial relation or through a new swapped edge (as in the bridge calculus). In the case of being accessible through the initial relation, the rule ensures that the edge used has not been deleted in the current model variant (as in the sabotage calculus). The $(\Box)$ rule, as in the sabotage calculus, works with all states accessible from n in the initial model variant, except when they have been made inaccessible in the current model variant. The $(\Box_2)$ rule, as in the bridge calculus, ensures that newly accessible states receive the formula φ.

The remaining (swapping) rules deserve more careful explanation. The three rules that handle formulas of the form $[sw]\varphi$ handle the case of swapping a reflexive edge, swapping an irreflexive edge that has never been swapped (nor its inverse), and swapping again an edge. $([sw])$ swaps reflexive edges, for which the S set does not need to be modified since swapping a reflexive edge leaves it unchanged. $([sw]_2)$ swaps irreflexive edges that have never been swapped before, i.e., usable edges (not in S) that are not in S^{-1}. This rule ensures that S is irreflexive ($n\dot{\neq}m$), asymmetric ($nm\dot{\notin}S^{-1}$) and that it does not contain two pairs of nominals that refer to the same edge in the induced model ($nm\dot{\notin}S$). Finally, $([sw]_3)$ traverses and swaps around edges of S^{-1}. If $n\dot{=}y$ is in the branch and $xy\in S$ then we swap again the link yx and end up at x. Hence it removes xy

from S and adds yx. This preserves the three properties of the set S (irreflexivity, asymmetry and no-redundant-names).

There is only one ($\langle sw\rangle$) rule but it handles three possibilities of satisfying a swap-diamond formula similarly to the rules for swap-box formulas. The ($\langle sw\rangle$) rule can satisfy a formula $(n, S) : \langle sw\rangle\varphi$ in three possible ways. First, through a reflexive edge, having φ true at n in the same model variant. In that case S remains unchanged. Or it satisfies it by adding an irreflexive edge to the initial relation ($\dot{R}nm$, $n\dot{\neq}m$), specifying that in the model variant S it is not removed nor is a new edge added by swapping ($nm\dot{\notin}(S\cup S^{-1})$), and then satisfying φ at m in the model variant $S \cup nm$. Finally, it can satisfy the antecedent formula by swapping again a swapped edge, updating S appropriately. The meaning of the last branch of this rule is to properly maintain the set S when an edge is swapped more than once. When an edge $xy \in S$ is swapped again, we update S by removing xy and adding yx, instead of adding a new pair of nominals.

Figure 7 shows the use of the tableau rules in an example.

Now we are going to prove completeness for the $\mathcal{ML}(\langle sw\rangle)$ calculus. Soundness can be shown similarly as for sabotage.

Lemma 14. *Let Θ be a saturated, open branch and φ a $\mathcal{ML}(\langle sw\rangle)$-formula. If $(n, S) : \varphi \in \Theta$ then $\mathcal{M}^{\Theta}_{S}, \bar{n} \models \varphi$.*

Proof. Let $(n, S) : \varphi \in \Theta$, we proceed by structural induction on φ. Propositional and Boolean cases are exactly the same that for $\mathcal{ML}(\langle sb\rangle)$.

- **$\Diamond\psi$:** We have two cases:
  1. $\dot{R}nm \in \Theta$, $nm\dot{\notin}S \in \Theta$ and $(m, S) : \psi \in \Theta$. Since $\dot{R}nm \in \Theta$, we have $(\bar{n}, \bar{m}) \in R^{\Theta}$. On the other hand, since $nm\dot{\notin}S \in \Theta$ and the branch is saturated and open, by Lemma 6, $\bar{n}\bar{m} \notin \bar{S}$. Then $\bar{n}\bar{m} \in R^{\Theta}_{S}$ and (by (Id)) $(\bar{m}, S) : \psi \in \Theta$. Hence, $\mathcal{M}^{\Theta}_{S}, \bar{n} \models \Diamond\psi$.
  2. $nm\dot{\in}S^{-1} \in \Theta$ and $(m, S) : \psi \in \Theta$. From the fist sentence, by Lemma 6, we have $\bar{n}\bar{m} \in \bar{S}$, hence $\bar{n}\bar{m} \in R^{\Theta}_{S}$. With the same argument that the previous item, we have $\mathcal{M}^{\Theta}_{S}, \bar{n} \models \Diamond\psi$.
- **$\langle sw\rangle\psi$:** ($\langle sw\rangle$) rule has three branches:
  1. $\dot{R}nn \in \Theta$ and $(n, S) \in \Theta$. In this case $\bar{n}\bar{n} \in R^{\Theta}_{S}$, and by ($Id$) $(\bar{n}, S) : \psi \in \Theta$, so we have $\mathcal{M}^{\Theta}_{S}, \bar{n} \models \langle sw\rangle\psi$.
  2. In the second branch, the following formulas belong to Θ: *a*) $\dot{R}nm$, *b*) $n\dot{\neq}m$, *c*) $nm\dot{\notin}(S \cup S^{-1})$ and *d*) $(m, S \ \cup nm) : \psi$. *b*) holds since we are not in the previous case. By *a*) and *c*) (and Lemma 6), we have $\bar{n}\bar{m} \in R^{\Theta}_{S}$. By ($Id$) and *d*), $(\bar{m}, S\cup nm) : \psi \in \Theta$. Hence, $\mathcal{M}^{\Theta}_{S}, \bar{n} \models \langle sw\rangle\psi$.
  3. In the third branch, there are $x, y \in W^{\Theta}$, such that $y\dot{=}n \in \Theta$ and $(x, S \setminus xy \cup yx) \in \Theta$.. Then $\bar{y}\dot{=}\bar{n} \in \Theta$ and by definition $\bar{y}\bar{x} \in R^{\Theta}_{S}$ $\otimes$. But, $(\bar{x}, S \setminus xy \cup yx) : \psi \in \Theta$, therefore $\mathcal{M}^{\Theta}_{S\setminus xy\cup yx}, \bar{x} \models \psi$. Then, since this last condition and $\otimes$, we have $\mathcal{M}^{\Theta}_{S}, \bar{n} \models \langle sw\rangle\psi$.
- **$\Box\psi$:** for all $m \in W^{\Theta}$ such that $\dot{R}nm$ and $nm\dot{\notin}S \in \Theta$, we have $(m, S) : \psi \in \Theta$. Because Θ is open and saturated, by Lemma 6 it holds that $\bar{n}\bar{m} \notin \bar{S}$, which implies $\bar{n}\bar{m} \in R^{\Theta}_{S}$. Otherwise, if $nm \in S^{-1}$, then also (by definition) $\bar{n}\bar{m} \in R^{\Theta}_{S}$. In both cases, we have $(\bar{m}, S) : \psi \in \Theta$. Hence, $\mathcal{M}^{\Theta}_{S}, \bar{n} \models \Box\psi$.

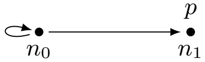
Example: Consider the formula $\neg p \wedge \langle sw\rangle\Diamond p$.

(1)	$(n_0,\emptyset) : \neg p \wedge \langle sw\rangle\Diamond p$		initial node
(2)	$(n_0,\emptyset) : \neg p$		$(\wedge)$ on (1)
(3)	$(n_0,\emptyset) : \langle sw\rangle\Diamond p$		
(4)	$\dot{R}n_0n_0, (n_0,\emptyset) : \Diamond p$	$\dot{R}n_0n_1, n_0\dot{\neq}n_1, (n_1,\{n_0n_1\}) : \Diamond p$	$(\langle sw\rangle)$ on (3)

Let us expand the left branch:

(5a)	$\dot{R}n_0n_1, n_0n_1\dot{\notin}\emptyset, (n_1,\emptyset) : p$		$(\Diamond)$ on (4)
(6a)	$n_0\doteq n_1$	$n_0\dot{\neq}n_1$	(ub)

The left branch closes by (Id) and $(\bot_{atom})$, while the right branch is fully expanded and open, with the following induced model:



Let us go back to line (4) and expand the right branch:

(5b)	$\dot{R}n_1n_2,\ n_1n_2\dot{\notin}\{n_0n_1\}$	$n_1n_2\dot{\in}\{n_1n_0\}$	$(\Diamond)$ on (4)
(6b)	$(n_2,\{n_0n_1\}) : p$	$(n_2,\{n_0n_1\}) : p$	

In the right branch, by $n_1n_2\dot{\in}\{n_1n_0\}$ we have $n_2\doteq n_0$. Then by (Id) and $(\bot_{atom})$, we have a clash. The left branch is open, and $n_1n_2\dot{\notin}\{n_0n_1\}$ is a notation for $n_0\dot{\neq}n_1 \vee n_1\dot{\neq}n_2$, with $n_0\dot{\neq}n_1$ already occurring in the branch (line (4), right branch).

(7b)	$n_0\doteq n_2$	$n_0\dot{\neq}n_2$	(ub)

Left branch closes by (Id) and $(\bot_{atom})$. Right branch:

(8b)	$n_1\doteq n_2$	$n_1\dot{\neq}n_2$	(ub)

Both branch are open and saturated and produce the following induced models:

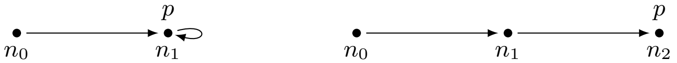


Fig. 7. Tableau example for $\mathcal{ML}(\langle sw\rangle)$

- $[sw]\psi$**:** the reflexive case is the same as for $\Box$. If we have in Θ that $\dot{R}nm$, $n\dot{\neq}m$ and $nm\dot{\notin}(S\cup S^{-1})$, then $\bar{n}\bar{m}\in R^{\Theta}_{S}$. Also we have $(\bar{m}, S\cup nm):\psi\in\Theta$. On the other hand, if $xy\dot{\in}S$ and $n\doteq y$ are both in Θ, (by definition) $\bar{y}\bar{x}\in R^{\Theta}_{S}$, and $(\bar{x}, S\backslash xy\cup yx){:}\psi\in\Theta$. With the three cases, we get $\mathcal{M}^{\Theta}_{S},\bar{n}\models[sw]\psi$. □

By the previous lemma we get:

Theorem 15 (Completeness). *If $\mathcal{T}(\varphi)$ is open, then φ is satisfiable.*

3 Global Relation-Changing Operators

In previous sections we considered only local operators that modify the model relation from the current state of evaluation. In particular, the sabotage and swap modalities traverse an existing accessibility relation from the current state. The bridge modality is local in the sense that it creates a new link also from the current state.

We now consider the global counterparts of these three modalities. These new versions can change the accessibility relation in any part of the model, and leave the evaluation state unchanged. One motivation to consider these global operators is, again, van Benthem's original sabotage operator [8], which is actually global.

The semantics of the three global operators is formally defined as follows:

$$\begin{array}{lll}\mathcal{M},w \models \langle gsb\rangle\varphi & \text{iff} & \text{for some } u,v \in W, \text{ s.t. } Ruv,\ \ \mathcal{M}^-_{uv},w \models \varphi\\ \mathcal{M},w \models \langle gbr\rangle\varphi & \text{iff} & \text{for some } u,v \in W \text{ s.t. } \neg Ruv,\ \mathcal{M}^+_{uv},w \models \varphi\\ \mathcal{M},w \models \langle gsw\rangle\varphi & \text{iff} & \text{for some } u,v \in W \text{ s.t. } Ruv,\ \ \ \mathcal{M}^*_{vu},w \models \varphi.\end{array}$$

Adapting the calculi presented in Section 2, we can obtain tableau methods for the global operations. For each logic, the corresponding ($\Diamond$) and ($\Box$) rules are the same ones as for its local version. One can easily verify that the rules for $\mathcal{ML}(\langle gsb\rangle)$ and $\mathcal{ML}(\langle gbr\rangle)$ in Figure 8 are direct adaptations of the rules for $\mathcal{ML}(\langle sb\rangle)$ and $\mathcal{ML}(\langle br\rangle)$. The rules for $\mathcal{ML}(\langle gsw\rangle)$ are shown in Figure 9). Notice that ($[gsw]_3$) and (the last branch produced by) ($\langle gsw\rangle$) are simpler than ($[sw]_3$) and ($\langle sw\rangle$). This is because swapping an already swapped edge in any place is a generalization of doing it only from the evaluation state.

$$\frac{(n,S):\langle gsb\rangle\varphi}{\begin{array}{c}\dot{R}pq\\ pq\mathrel{\dot{\notin}}S\\ (n,S\cup pq):\varphi\end{array}}\;(\langle gsb\rangle)^1 \qquad\qquad \frac{(n,B):\langle gbr\rangle\varphi}{\begin{array}{c}pq\mathrel{\dot{\notin}}B\\ (n,B\cup pq):\varphi\end{array}}\;(\langle gbr\rangle)^1$$

$$\frac{\begin{array}{c}(n,S):[gsb]\varphi\\ \dot{R}pq\\ pq\mathrel{\dot{\notin}}S\end{array}}{(n,S\cup pq):\varphi}\;([gsb]) \qquad\qquad \frac{\begin{array}{c}(n,B):[gbr]\varphi\\ pq\mathrel{\dot{\notin}}B\end{array}}{(n,B\cup pq):\varphi \;\mid\; \dot{R}pq}\;([gbr])$$

[1] p and q are new to the branch.

Fig. 8. Tableau rules for $\mathcal{ML}(\langle gsb\rangle)$ and $\mathcal{ML}(\langle gbr\rangle)$

The resulting calculi are sound and complete. The complexity for the satisfiability of these logics is still open but we conjecture they are undecidable (a close variant of $\mathcal{ML}(\langle gsb\rangle)$ is undecidable [6]). Applying similar arguments as for the local operators, it is possible to at least enforce infinite models.

$$\frac{\begin{array}{c}(n,S):[gsw]\varphi\\ \dot{R}pp\end{array}}{(n,S):\varphi}\,([gsw]) \qquad \frac{\begin{array}{c}(n,S):[gsw]\varphi\\ \dot{R}pq\\ p\dot{\neq}q\\ pq\dot{\notin}(S\cup S^{-1})\end{array}}{(n,S{\cup}pq):\varphi}\,([gsw]_2) \qquad \frac{\begin{array}{c}(n,S):[gsw]\varphi\\ xy\in S\end{array}}{(n,S{\setminus}xy{\cup}yx):\varphi}\,([gsw]_3)$$

$$\frac{(n,S):\langle gsw\rangle\varphi}{\begin{array}{c|c|l}\dot{R}pp & \dot{R}pq & \bigvee_{xy\in S}(n,S{\setminus}xy{\cup}yx){:}\varphi\\ (n,S):\varphi & p\dot{\neq}q & \\ & pq\dot{\notin}(S{\cup}S^{-1}) & \\ & (n,S{\cup}pq):\varphi & \end{array}}\,(\langle gsw\rangle)^1$$

[1] p and q are new to the branch.

Fig. 9. Tableau rules for $\mathcal{ML}(\langle gsw\rangle)$

4 Ending Remarks

In this article we considered a number of dynamic operators which can add, delete and swap edges in the accessibility relation, both locally and globally. We introduced sound and complete tableau procedures for all of them to check satisfiability.

A natural question is whether it is possible to combine these calculi into a unique calculus that would support modal logic equipped with all the dynamic operators at once. We can easily obtain local-global combinations of calculi for operators of the same kind: $\mathcal{ML}(\langle sb\rangle,\langle gsb\rangle)$, $\mathcal{ML}(\langle br\rangle,\langle gbr\rangle)$ and $\mathcal{ML}(\langle sw\rangle,\langle gsw\rangle)$, by combining the corresponding rules from Section 2 and Section 3. However, further combination seems to require deep changes since every kind of dynamic logic (sabotage, bridge, swap) requires distinct rules for the connectors $\Diamond$ and $\Box$.

As can be seen from their corresponding calculi, the logics presented here involve equality reasoning on named states. They are actually related to hybrid logics [5,1]. In particular $\mathcal{ML}(\langle sw\rangle)$ is strictly less expressive than $\mathcal{H}(:,\downarrow)$ [3]. The same can be shown about $\mathcal{ML}(\langle sb\rangle)$ and $\mathcal{H}(:,\downarrow)$. Let $S\subseteq \mathsf{NOM}^2$ and $x',y'\in \mathsf{NOM}$. Define $(\ )'_S$, a translation from formulas of $\mathcal{ML}(\langle sb\rangle)$ to formulas of $\mathcal{H}(:,\downarrow)$ as (for the non-trivial cases):

$$\begin{aligned}(\Diamond\varphi)'_S &= \downarrow x'.\Diamond\downarrow y'.(\neg\bigvee_{xy\in S}(x'{:}x\wedge y'{:}y) \quad\wedge\quad (\varphi)'_S)\\ (\langle sb\rangle\varphi)'_S &= \downarrow x'.\Diamond\downarrow y'.(\neg\bigvee_{xy\in S}(x'{:}x\wedge y'{:}y) \quad\wedge\quad (\varphi)'_{S\cup x'y'})\end{aligned}$$

where x' and y' are nominals that do not appear in S. With this translation it holds that for any formula φ of $\mathcal{ML}(\langle sb\rangle)$ and pointed model $\mathcal{M},w$, we have $\mathcal{M},w\models\varphi$ iff $\mathcal{M},w\models(\varphi)'_S$. On the other hand, translation for the four remaining logics involve the global modality E.

All of the logics we considered can force infinite models. As a result, the tableau calculi not necessarily terminate on all inputs, given that they do not implement any kind of loop checking. Our ongoing research aims to establish the undecidability of all the presented logics using techniques from [3], showing in this way that non-termination is unavoidable.

As future work, we plan to investigate constructive interpolation results in hybrid versions of the logics we presented here.

Acknowledgments. This work was partially supported by grants ANPCyT-PICT-2008-306, ANPCyT-PICT-2010-688, the FP7-PEOPLE-2011-IRSES Project "Mobility between Europe and Argentina applying Logics to Systems" (MEALS) and the Laboratoire International Associé "INFINIS".

References

1. Areces, C., ten Cate, B.: Hybrid logics. In: Blackburn, P., Wolter, F., van Benthem, J. (eds.) Handbook of Modal Logics, pp. 821–868. Elsevier (2006)
2. Areces, C., Fervari, R., Hoffmann, G.: Moving arrows and four model checking results. In: Ong, L., de Queiroz, R. (eds.) WoLLIC 2012. LNCS, vol. 7456, pp. 142–153. Springer, Heidelberg (2012)
3. Areces, C., Fervari, R., Hoffmann, G.: Swap logic. To appear in the Logic Journal of IGPL (2013)
4. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic, Cambridge Tracts in Theoretical Computer Science, vol. 53. Cambridge University Press, Cambridge (2001)
5. Blackburn, P., Seligman, J.: Hybrid languages. Journal of Logic, Language and Information 4, 251–272 (1995)
6. Löding, C., Rohde, P.: Model checking and satisfiability for sabotage modal logic. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 302–313. Springer, Heidelberg (2003)
7. Schmidt, R.A., Tishkovsky, D.: Using tableau to decide expressive description logics with role negation. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 438–451. Springer, Heidelberg (2007)
8. van Benthem, J.: An essay on sabotage and obstruction. In: Hutter, D., Stephan, W. (eds.) Mechanizing Mathematical Reasoning. LNCS (LNAI), vol. 2605, pp. 268–276. Springer, Heidelberg (2005)

Computing Minimal Models Modulo Subset-Simulation for Modal Logics[*]

Fabio Papacchini and Renate A. Schmidt

The University of Manchester, UK
{papacchf,schmidt}@cs.man.ac.uk

Abstract. In this paper we propose a novel minimality criterion for models of modal logics based on a variation of the notion of simulation, called subset-simulation. We present a minimal model sound and complete tableau calculus for the generation of this new kind of minimal models for the multi-modal logic $\mathbf{K}_{(m)}$, and we discuss extensions to cover more expressive logics. The generation of minimal models is performed incrementally by using a minimality test to close branches representing non-minimal models, or to update the set of minimal models. Subset-simulation minimal models have the advantage that they are semantically more natural than models obtained by using syntactic minimality criteria.

1 Introduction

For fault analysis, verification of systems and validation of the logical formalisation of an application, model generation methods are useful for finding counterexamples as a means of debugging [14]. Models can be generated using tableau methods. For example, Smullyan-type labelled tableau calculi can be used to generate the essential parts of any model. However, even for the most well-behaved, decidable logics, in general, there are uncountably many different models for satisfiable formulae and models can be very large. The import of Herbrand's theorem is that we can restrict attention to the class of Herbrand models, because they are kinds of canonical models sufficient for showing soundness and completeness of many deductive systems. For the purposes of model generation, the class of Herbrand models has the advantage that it can be ordered by the subset relation. It is thus possible to focus on generating models minimal under this ordering. Generating minimal Herbrand models for classical logics has been studied in [4,12] and for modal logics in [13]. For the modal logics **K**, **KT**, **KB**, **KTB** it has been shown minimal Herbrand models are finite [13], but for other extensions of **K** minimal Herbrand models are in general infinite.

By contrast, domain minimal models are finite for all logics with the finite model property. Another possibility therefore is to focus on the generation of models with minimised domains. Domain minimal models, however, tend to be counter-intuitive for verification and debugging purposes because too many

[*] The first author is supported by an EPSRC EU Doctoral Training Award.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 279–294, 2013.

Table 1. Modalities and their corresponding frame conditions

$[R_i]$	Axiom	Frame condition
K		
T	$[R_i]p \rightarrow p$	reflexivity
B	$p \rightarrow [R_i]\langle R_i\rangle p$	symmetry
D	$[R_i]p \rightarrow \langle R_i\rangle p$	seriality
4	$[R_i]p \rightarrow [R_i][R_i]p$	transitivity
5	$\langle R_i\rangle p \rightarrow [R_i]\langle R_i\rangle p$	Euclideanness

worlds are collapsed to a single world. For instance, in a modal logic with doxastic modalities there are models in which belief states (those in the image of the belief relation) are identified and reflexive loops created. In most formalisations the belief relation is however not reflexive. This means there is a need to find classes of models better suited for debugging purposes.

As Herbrand models are too large and domain minimal models are too small, in this paper we study subset-simulation minimal models as a middle ground between the two. Subset-simulation is a relationship between models based on a variation of the notion of simulation [5,7,8]. Being applied directly on the graph representation of models means subset-simulation minimality preserves the semantics in minimal models, and is suitable for a large number of non-classical logics. It also results in more natural and intuitive minimal models than minimal Herbrand models and domain minimal models (Section 3).

We present a tableau calculus designed to generate subset-simulation minimal models for the multi-modal logic $\mathbf{K}_{(m)}$ in Section 5. The tableau is minimal model complete, but it is not minimal model sound. That is, it generates all minimal models, but also non-minimal models are generated. Section 6 shows how the calculus can be extended with a minimality test, called subset-simulation test, in order to generate only minimal models and achieve minimal model soundness. The resulting approach iteratively computes exactly the models minimal modulo subset-simulation by updating the set of minimal models as the derivation proceeds. Although the calculus we present is for the multi-modal logic $\mathbf{K}_{(m)}$, extensions to cover more expressive logics are easy to obtain. We conclude the paper with a discussion of possible extensions of the calculus and remarks on implementation (Section 7).

2 Preliminaries

We work with modal formulae of propositional multi-modal logic $\mathbf{K}_{(m)}$ possibly extended with universal modalities or a subset of the well-known axioms **T**, **B**, **D**, **4**, and **5**. Table 1 lists the axioms and their semantic meaning.

A *modal formula* is a formula of the form $\top$, $\bot$, p_i, $\neg\phi$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\langle R_i\rangle\phi$, $[R_i]\phi$, $[\mathcal{U}]\phi$, $\langle\mathcal{U}\rangle\phi$, where $\top$ and $\bot$ are two nullary logical operators for, respectively, true and false; p_i is a propositional symbol; $\neg$, $\wedge$, $\vee$, $\langle R_i\rangle$, $[R_i]$ are, respectively, the logical operators negation, conjunction, disjunction, diamond and box; $[\mathcal{U}]$ and $\langle\mathcal{U}\rangle$ are universal modalities; and ϕ_1, ϕ_2, ϕ are modal formulae.

We adopt the standard semantics of modal formulae known as Kripke semantics. A *frame* for multi-modal logics is a tuple $(W, \mathcal{R})$, where W is a non-empty set of worlds and $\mathcal{R} = \{R_1, \ldots, R_n\}$ is a set of accessibility relations over W. An *interpretation* $\mathcal{I}$ is a tuple $(W, \mathcal{R}, V)$ composed of a frame and an interpretation function V that assigns to each world $u \in W$ a set propositional symbols meaning that such propositional symbols hold in u. Given an interpretation $\mathcal{I} = (W, \mathcal{R}, V)$ and a world $u \in W$, truth of a modal formula ϕ is inductively defined as follows.

$$\begin{array}{ll}
\mathcal{I}, u \not\models \bot & \mathcal{I}, u \models \top \\
\mathcal{I}, u \models p_i & \text{ iff } p_i \in V(u) \\
\mathcal{I}, u \models \neg\phi & \text{ iff } \mathcal{I}, u \not\models \phi \\
\mathcal{I}, u \models \phi_1 \vee \phi_2 & \text{ iff } \mathcal{I}, u \models \phi_1 \text{ or } \mathcal{I}, u \models \phi_2 \\
\mathcal{I}, u \models \phi_1 \wedge \phi_2 & \text{ iff } \mathcal{I}, u \models \phi_1 \text{ and } \mathcal{I}, u \models \phi_2 \\
\mathcal{I}, u \models [R_i]\phi & \text{ iff for every } v \in W \text{ if } (u, v) \in R_i \text{ then } \mathcal{I}, v \models \phi \\
\mathcal{I}, u \models \langle R_i\rangle\phi & \text{ iff there is a } v \in W \text{ such that } (u, v) \in R_i \text{ and } \mathcal{I}, v \models \phi \\
\mathcal{I}, u \models [\mathcal{U}]\phi & \text{ iff for every } v \in W\ \mathcal{I}, v \models \phi \\
\mathcal{I}, u \models \langle\mathcal{U}\rangle\phi & \text{ iff there is a } v \in W \text{ such that } \mathcal{I}, v \models \phi
\end{array}$$

Given an interpretation $\mathcal{I}$, a world u and a modal formula ϕ, if $\mathcal{I}, u \models \phi$ holds, then $\mathcal{I}$ is a *model* of ϕ.

3 Subset-Simulation as Minimality Criterion

Subset-simulation is a variation of the notion of simulation [8,7,5], and is known from [1,11], where it is used for the description logic $\mathcal{EL}$ and is simply called simulation. As we use both simulation and its variation, we decided to call the latter subset-simulation.

Let $M = (W, \mathcal{R}, V)$ and $M' = (W', \mathcal{R}', V')$ be two models of a modal formula ϕ. A *simulation* is a binary relation $S \subseteq W \times W'$ such that for any two worlds $u \in W$ and $u' \in W'$, if uSu' then the following hold.

- $V(u) = V'(u')$ and
- if uRv for some $R \in \mathcal{R}$, then there exists a $v' \in W'$ such that $R \in \mathcal{R}'$, $u'Rv'$, and vSv'.

Let $M = (W, \mathcal{R}, V)$ and $M' = (W', \mathcal{R}', V')$ be two models of a modal formula ϕ. A *subset-simulation* is a binary relation $S_\subseteq \subseteq W \times W'$ such that for any two worlds $u \in W$ and $u' \in W'$, if $uS_\subseteq u'$ then the following hold.

- $V(u) \subseteq V'(u')$ and
- if uRv for some $R \in \mathcal{R}$, then there exists a $v' \in W'$ such that $R \in \mathcal{R}'$, $u'Rv'$ and $vS_\subseteq v'$.

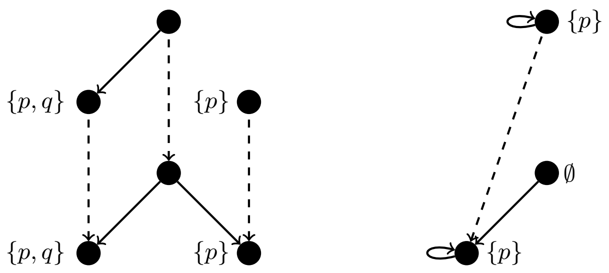


Fig. 1. Simulations between symmetric models modulo subset-simulation

If S is such that for all $u \in W$ there is at least one $u' \in W'$ such that uSu', then we call S a *full (subset-)simulation* from M to M'. We say a (subset-)simulation S is a *maximal (subset-)simulation* if there is no other (subset-)simulation $S' \neq S$ such that $S \subset S'$. Given two models M and M', if there is a full (subset-)simulation S from M to M', we say that M' *(subset-)simulates* M, or M *is (subset-)simulated* by M'.

In this paper we are only interested in full and maximal (subset-)simulations. For this reason, when we refer to (subset-)simulations we mean full and maximal (subset-)simulations. Where ambiguous, we explicitly state what kind of (subset-)simulation we mean.

Subset-simulation has properties that allow us to use it to define a minimality criterion for modal logic models. Specifically, subset-simulation is a reflexive and transitive relation on models. Hence, it forms a preorder on models. This means that we can consider as minimal models all the models that are minimal with respect to subset-simulation. As subset-simulation is not anti-symmetric (which means it is not a partial order), it is possible for models to form a symmetry class. A *symmetry class* is a set of models that subset-simulate each other. This may result in minimal models belonging to large symmetry classes, and therefore also a large number of minimal models. To overcome this problem, we aim to be more restrictive by using also the notion of simulation within a symmetry class of minimal models.

Let M be a model of a modal formula ϕ. M is *minimal modulo subset-simulation* iff for any other model M' of ϕ, if M subset-simulates M', then M' subset-simulates M and either there is no simulation relationship between M and M', or M' simulates M.

The use of a simulation check within a symmetry class allows us to recognise bisimilar models, models that are embedded in other models, and to impose an extra ordering over symmetric models. Figure 1 shows two examples of what kinds of minimal models are excluded due to the use of simulation. The two models on the right belong to a symmetry class, and the two models on the left belong to a different symmetry class (that is, the models on the right subset-simulate each other, and the models on the left subset-simulate each other). The dashed lines in the figure represent the simulation relationships between

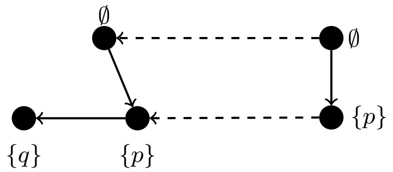


Fig. 2. Minimality modulo subset-simulation vs. minimal Herbrand models

the models. As the models at the bottom simulates the models at the top, only the two models at the top are minimal modulo subset-simulation.

Models minimal modulo subset-simulation have interesting properties. First, the interpretation function is minimal with respect to a given frame. This means that given a set of worlds and the accessibility relations between them, the interpretation function assigns to each world the minimal number of propositional variables such that the resulting interpretation is a model for a given formula. Second, as subset-simulation is directly based on the graph structure of models, the minimality criterion is able to discern models semantically, thus avoiding semantically redundant minimal models (in opposition to the minimal Herbrand models criterion). In other words, the criterion is able to compare models having distinct domains by comparing directly the labelling functions and the accessibility relations. Being based on the graph structure of models makes minimality modulo subset-simulation a criterion suitable for a large number of modal logics. Finally, subset-simulation gives priority to finite loop-free models, meaning that usually models minimal modulo subset-simulation are not domain minimal.

We conclude this section with two examples of models minimal modulo subset-simulation in order to compare the notion with other minimality criteria. The first example shows that the new minimality criterion does not suffer the syntactic restriction that affects Herbrand models. For lack of space we cannot give the formal definition of modal Herbrand models for which we refer to [13]. Even though there are a few differences, it might help to think of them as the Herbrand models of the translation of a modal formula into a first-order formula. Let us consider the modal formula $\phi = \langle R_1 \rangle p \vee \langle R_1 \rangle (p \wedge \langle R_1 \rangle q)$. The minimal Herbrand models of ϕ are shown in Figure 2. As can be seen, the model on the right is completely embedded in the model on the left. Due to the syntactic restrictions of Herbrand models, however, it is not possible to recognise this relation between models, and the method proposed in [13] would consider both the as being minimal models. By contrast, subset-simulation minimality considers only the model on the right as minimal because it is subset-simulated by the other model, but not the other way around.

The second example shows that models minimal modulo subset-simulation are more natural than domain minimal models. Consider the formula $\phi = \langle has\_father \rangle doctor$. Figure 3 shows two models that satisfy ϕ. The left model in the figure is the domain minimal model, and the right model is the model minimal modulo subset-simulation. In the domain minimal model ϕ is satisfied

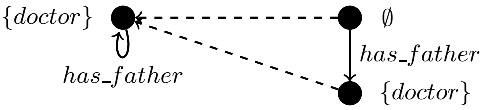


Fig. 3. Minimality modulo subset-simulation vs. domain minimality

by creating a loop, meaning that there is a person who is their own father and who is a doctor. Even though such a model satisfies ϕ, it does not reflect our intuition of the *has_father* relation. This problem is avoided in the model minimal modulo subset-simulation, where a new successor is required, thus ensuring that a person is not their own father. Admittedly this is a simple example, but it illustrates problems avoided for relations where reflexivity is counter-intuitive. Our main point is avoiding loops in models if they are not necessary for the finiteness of the model, and we only create loops containing the least positive information by minimising the interpretation function when necessary.

4 Computing Subset-Simulation between Models

Even though not concerned with modal logic models, the paper [8] presents algorithms for computing simulations between graphs. One of the algorithms presented in [8] computes maximal self-simulation, which is the maximal simulation between a graph and itself. This algorithm can be modified for computing full and maximal subset-simulation between two models of a modal formula ϕ.

Figure 4 shows the pseudo-code of the algorithm that takes as input two models $M = (W, \mathcal{R}, V)$ and $M' = (W', \mathcal{R}', V')$, and returns the full and maximal subset-simulation from M to M' or the empty set, if there is no full subset-simulation. The following variables and functions are used in the algorithm.

- $outrel(u)$ returns the set of outgoing accessibility relations from u.
- $sim(u)$ represents the set of worlds in W' that are subset-similar to u.
- $post(R_i, u)$ returns all the R_i-successors of u.
- $pre(R_i, u)$ returns all the R_i-predecessors of u.
- $pre(R_i, W)$ is the union of the sets resulting by the application of the *pre* function with respect to the relation R_i to all the elements of W, that is, $pre(R_i, W) = \bigcup_{u \in W} pre(R_i, u)$.

The basic idea behind the algorithm is that subset-simulation is computed by overestimating the possible subset-simulation, and then pruning false guesses by checking the second property of subset-simulation. To obtain the algorithm in Figure 4 from the algorithm presented in [8], the following must be taken into consideration. First, if a full subset-simulation does not exist, then the empty set is returned. Second, the input is composed of two different models, this is because we are not interested in self subset-simulations. Reflexive edges need to be correctly handled. The two graphs have more than one accessibility relation.

```
for all $v \in W$ do
    $sim(v) \leftarrow \{u \in W' \mid V(v) \subseteq V'(u)$ and $outrel(v) \subseteq outrel(u)\}$
    if $sim(v) = \emptyset$ then
        return $\emptyset$
for all $R_i \in \mathcal{R}$ do
    for all $v \in W$ do
        $remove(v) \leftarrow pre(R_i, W') \setminus pre(R_i, sim(v))$
    while there is a vertex $v \in W$ s.t. $remove(v) \neq \emptyset$ do
        $aux\_remove\_v \leftarrow \emptyset$
        for all $u \in pre(R_i, v)$ do
            for all $w \in remove(v)$ do
                if $w \in sim(u)$ then
                    $sim(u) \leftarrow sim(u) \setminus \{w\}$
                    for all $w' \in pre(R_i, w)$ do
                        if $post(R_i, w') \cap sim(u) = \emptyset$ then
                            if $u = v$ then
                                $aux\_remove\_v \leftarrow aux\_remove\_v \cup \{w'\}$
                            else
                                $remove(u) \leftarrow remove(u) \cup \{w'\}$
            if $sim(u) = \emptyset$ then
                return $\emptyset$
        $remove(v) \leftarrow aux\_remove\_v$
return $\{(u, v) \mid u \in W$ and $v \in sim(u)\}$
```

Fig. 4. Pseudo-code for computing full, maximal subset-simulation between two models

This last point is the reason why the algorithm loops over the set of accessibility relations and refines the subset-simulation by incrementally computing the intersection of the subset-simulation with respect to a single accessibility relation. For reason of space we omit a more detailed description of the algorithm.

Soundness of computing the subset-simulation in this way is a consequence of the following theorem.

Theorem 1. *Let $M = (W, \mathcal{R}, V)$ and $M' = (W', \mathcal{R}', V')$ be two models of a modal formula ϕ such that M is subset-simulated by M'. The full and maximal subset simulation $S_{\subseteq}$ can be computed as the intersection of all the full and maximal subset-simulations with respect to each single accessibility relation $R \in \mathcal{R}$.*

5 Tableau Calculus

We present a generic tableau calculus for the generation of minimal models modulo subset-simulation for the multi-modal logic $\mathbf{K}_{(m)}$. The calculus is generic in the sense that it can be easily extended to more expressive modal logics. Such extensions are discussed in Section 7.

The input of the calculus is a modal formula in negation normal form labelled by an initial world u. Transformation to negation normal form is not essential,

Table 2. Rules of the basic tableau calculus

$$(\alpha)\ \frac{u:(\phi_1\wedge\ldots\wedge\phi_n)\vee\Phi^+_\alpha}{\begin{array}{c}u:\phi_1\vee\Phi^+_\alpha\\ \vdots\\ u:\phi_n\vee\Phi^+_\alpha\end{array}} \qquad (\Box)\ \frac{(u,v):R_i \quad u:[R_i]\phi}{v:\phi}$$

$$(\beta)\ \frac{u:\mathcal{A}\vee\Phi^+}{\begin{array}{l|l} u:\mathcal{A} & u:\Phi^+\\ u:\mathit{neg}(\Phi^+) & \end{array}}$$

where $\mathcal{A}$ is of the form $\langle R_i\rangle\phi$, $[R_i]\phi$, or p_i, and $\mathit{neg}(\Phi^+)=\neg p_1\wedge\ldots\wedge\neg p_n$, where each p_i is a disjunct of Φ^+.

$$(\Diamond)\ \frac{u:\langle R_i\rangle\phi}{\begin{array}{l|l|l|l}(u,u_1):R_i & \ldots & (u,u_n):R_i & (u,v):R_i\\ u_1:\phi & & u_n:\phi & v:\phi\end{array}}$$

where each u_i appears on the branch, and v is fresh.

$$(SBR)\ \frac{u:p_1,\ldots,u:p_n \quad u:\neg p_1\vee\ldots\vee\neg p_n\vee\Phi^+_\alpha}{u:\Phi^+_\alpha}$$

but it simplifies the presentation. It also means that there is no need for preprocessing before applying the calculus, and it allows us to reduce the number of rules in the calculus.

In the calculus, disjunctions and conjunctions are assumed to be flattened for example, we write $\phi_1\vee\phi_2\vee\phi_3$ instead of $\phi_1\vee(\phi_2\vee\phi_3)$. By $\mathcal{A}$ we mean a modal formula of the form p_i, $\langle R_i\rangle\phi$ or $[R_i]\phi$. We use Φ^+ to denote a non-empty disjunction, where all disjuncts are of the form $\mathcal{A}$, and Φ^+_α to denote a possibly empty disjunction where all disjuncts are of the form $\mathcal{A}$ or are conjunctions. By $\mathit{neg}(\Phi^+)$ we mean the conjunction $\neg p_1\wedge\ldots\wedge\neg p_n$, where the p_i are all the positive propositional variables appearing as disjuncts of Φ^+. If Φ^+ does not contain any p_i, then $\mathit{neg}(\Phi^+)=\top$.

Table 2 presents the rules of the calculus for the multi-modal logic $\mathbf{K}_{(m)}$. A branch $\mathcal{B}$ of the tableau is a sequence $N_0,N_1,\ldots,N_i$ of sets of formulae of the form $u:\phi$ or $(u,v):R$, where $N_0=\{u:\phi\}$ and ϕ is the input formula. Given an input formula $u:\phi$, the rules of the calculus are exhaustively applied. At most one rule is applied to any formula appearing as the main premise, where the main premise of multi-premises rules is the premise on the right. For fairness, each instance of a rule application is applied exactly once. Each rule application extends the current branch. That is, a rule applied to a formula in the set N_i extends the branch with the set N_{i+1}, where N_{i+1} is the set N_i plus the conclusions of the applied rule. Given an open branch $\mathcal{B}$, a model $M=(W,\mathcal{R},V)$ can be extracted from $\mathcal{B}$ as follows. The domain W is the set of all the labels in $\mathcal{B}$, the accessibility relations are composed of all the instances $(u,v)\in R_i$ in $\mathcal{B}$, the interpretation function V is such that $V(u)=\{p_i\mid u:p_i\in\mathcal{B}\}$.

Even though the input formula is in negation normal form, the calculus can be thought of as a calculus for formulae in clausal form. This is achieved by the (α) rule that not only deals with conjunctions, but also performs lazy clausification. If such a lazy clausification is performed in a clever way, for example, by using a good heuristic for choosing the right conjunction to expand, it can result in the reduction of inferences due to the implicit restriction of Φ_α^+ in the premise of the rule. For instance, let us assume that the premise is $u : (p_1 \wedge p_2) \vee (\neg p_3 \wedge \neg p_4)$. If the (α) rule is applied to the first conjunction, it results in the two modal formulae $u : p_1 \vee (\neg p_3 \wedge \neg p_4)$ and $u : p_2 \vee (\neg p_3 \wedge \neg p_4)$. The (α) rule is again applicable to both of them. If instead, the (α) rule is applied to the other conjunction first, the resulting formulae are $u : \neg p_3 \vee (p_1 \wedge p_2)$ and $u : \neg p_4 \vee (p_1 \wedge p_2)$, and the (α) rule is not applicable to any of them.

The $(\Box)$ rule is the common rule for box formulae, and simply expands formulae in the scope of a box modality as required by the semantics of box formulae.

The (β) rule is one of the two branching rules of the calculus. Its purpose is to branch over disjunctions without any negated propositional variables, and to close the left branch if it is not minimal. This latter point is achieved by the use of a limited form of complement splitting (a more common use of complement splitting can be found in [4]). The reason why complement splitting is applied only on positive propositional variables is that the negation of diamond formulae or box formulae would result in new modal formulae (specifically, box formulae and diamond formulae) that can compromise the minimality of the resulting model. For example, let us assume that the (β) rule is applied to $u : p \vee [R_1]q$. If the complement $\langle R_1\rangle\neg q$ of $[R_1]q$ would have been added to the left branch, the left branch would still be open, and the resulting model would still be a model for the original formula, but the newly introduced diamond formula would generate unnecessary information. The resulting model would not be minimal. A similar example can be given for the case of the negation of diamond formulae.

The $(\Diamond)$ rule is the expansion rule for diamond formulae. As it can lead to the expansion of a diamond formula in all possible worlds plus a fresh one, it is an expensive rule. It is, however, required to achieve minimal model completeness. This rule is known from literature, for example [9,10,3]. It is worth pointing out that the $(\Diamond)$ rule in general does not guarantee termination for the purpose of minimal model generation, but it ensures termination in case we are only interested in checking the satisfiability of a modal formula belonging to a logic with the finite model property. The termination issue for minimal model generation does not affect the multi-modal logic $\mathbf{K}_{(m)}$, but it has to be taken into consideration when generalising to more expressive logics.

Finally, the (SBR) rule is a selection-based resolution rule. It can be seen as a weaker version of the (SBR) rule in [13], the $PUHR$ rule in [4], or the hypertableau rule in [2]. The aim of this rule is twofold. First, it provides the closure rule of the calculus, because atomic closure is sufficient. Second, it allows to remove negative information (that is, all negative propositional variables) from a disjunction. The reason behind the (SBR) rule is that if a disjunction contains negative information (that is, at least one negated propositional variable) that is not in conflict with any formula on the branch, then any expansion of such a disjunction results in either the minimal model, where the disjunction is true due to the negative information, or in a non-minimal model. Hence, there is no advantage in expanding a disjunction as long as it is not possible to remove all the negative information from it. The (SBR) rule is the reason why other rules, specifically the (β) rule and the (α) rule, can be applied only to disjunctions of the form Φ^+ or Φ^+_α. This decreases the number of required inferences.

The calculus presented so far does not yet constitute the full method for the generation of models minimal modulo subset-simulation, but is a starting point for it.

Theorem 2. *The tableau calculus is refutationally sound and complete for* $\mathbf{K}_{(m)}$.

For lack of space we do not provide a formal proof, but the calculus does not differ much from known calculi. All the rules have already been applied in other calculi, or are sound variations of common rules. The main differences are due to variations of rules in order to not expand formulae that are already minimally satisfied, for example, the restrictions due to Φ^+ or Φ^+_α.

Theorem 3. *The tableau calculus is subset-simulation minimal model complete. That is, it generates all models minimal modulo subset-simulation.*

Proof. Suppose $M, u \models \phi$, where $M = (W, \mathcal{R}, V)$ is a model minimal modulo subset-simulation, $u \in W$ and ϕ is a modal formula. We first show that the tableau having as input $u : \phi$ has an open, fully expanded branch $\mathcal{B} = N_0, \ldots, N_i, \ldots$, where $N_0 = \{u : \phi\}$ and for all $i \geq 0$ the following holds: $M \models N_i$ implies $M \models N_{i+1}$, where $M \models N_i$ means that for each formula $u : \phi \in N_i$ we have that $M, h(u) \models \phi$, where h is a function mapping labels in N_i to domain elements in M such that $h(u) = u$ for the starting label u. Suppose N_{i+1} is obtained from N_i by the application of a rule ρ. We consider several cases.

ρ is the $(\Box)$ rule. This means the expanded formula is a labelled box formula $u : [R_i]\phi'$, and $(u, v) : R_i$ is in N_i for some v. As $M \models N_i$, we have that $M, h(u) \models [R_i]\phi'$ and $(h(u), h(v)) \in R_i$. This implies $M, h(v) \models \phi'$. That is, M is a model for the conclusion of the application of the $(\Box)$ rule.

ρ is the (α) rule. This means that the expanded formula is a labelled disjunction, where at least one disjunct ϕ_α is a conjunction. As $M \models N_i$, we have that $M, h(u) \models \phi$. This implies that $M, h(u) \models \phi'_\alpha$, where ϕ'_α is the result of distributing the conjunction of ϕ_α over ϕ. Hence, M is a model for all the conjuncts of ϕ'_α. That is, M is a model for the conclusions of the application of the (α) rule.

ρ is the $(\Diamond)$ rule. This means that the expanded formula is a labelled diamond formula, let us say $u : \langle R_i \rangle \phi'$. As $M \models N_i$, we have that $M, h(u) \models \langle R_i \rangle \phi'$. This implies that there exists an R_i-successor v of $h(u)$ such that $M, v \models \phi'$. If there is no w in N_i such that $h(w) = v$, then we choose the right-most conclusion of the $(\Diamond)$ rule and let $h(w) = v$. If there is already a world w in N_i such that $h(w) = v$, then choose the conclusion where w is used as witness.

ρ is the (β) rule. This means that the expanded formula is a labelled disjunction where the disjuncts are propositional variables, diamond formulae or box formulae. As $M \models N_i$, we have that $M, h(u) \models \phi$. This implies that M satisfies at least one of the disjuncts. Suppose $\mathcal{A}$ is one such disjunct and Φ^+ is the remaining part of the disjunction. Assume $\mathcal{A}$ is expanded in the left branch, then two cases are possible. First, $M, h(u) \models neg(\Phi^+)$, that is, M is a model for all the conclusions in the left branch. Second, $M, h(u) \not\models neg(\Phi^+)$. That is, there is a propositional variable p_i that appears as disjunct in Φ^+ such that $M, h(u) \models p_i$. This means that $u : \phi$ is already satisfied because p_i is satisfied, that is, one of the disjunct of Φ^+ is satisfied. Hence, $M, h(u) \models \Phi^+$ and the correct expansion of N_i is the right branch of the (β) rule.

ρ is the (SBR) rule. This means that the expanded formula $u : \phi$ is of the form $u : \neg p_1 \vee \ldots \vee \neg p_n \vee \phi'$ and that $u : p_1, \ldots, u : p_n$ appear in N_i. As $M \models N_i$, we have that $M, h(u) \models \phi$ and $M, h(u) \models p_i$ for all i within $1 \leq i \leq n$. This implies that $M, h(u) \models \phi$ iff $M, h(u) \models \phi'$. That is, M is a model for the conclusion of the (SBR) rule.

This proves by induction that there is a branch $\mathcal{B}$ validated by M.

It remains to show that the model $M' = (W', \mathcal{R}', V')$ extracted from $\mathcal{B}$ is equivalent to M. From the construction of the branch, the domain of M' is such that $W'_h \subseteq W$, where $W'_h = \{v \mid h(u) = v \text{ for all } u \in W'\}$. This is, because the starting node u belongs to both W' and W, only applications of the $(\Diamond)$ rule create worlds, and these are mapped by following what holds in the minimal model M. The same reasoning is also applicable for the set of accessibility relations.

The interpretation function V' is such that for all $u : p_i \in \mathcal{B}$, $p_i \in V'(u)$. This implies that for all $u \in W'$ we have that $V'(u) \subseteq V(h(u))$. This is because $M \models N_i$ for all i and, specifically, $M, h(u) \models p_i$ for all $u : p_i \in \mathcal{B}$.

From these observations it follows that M' is either smaller (containing fewer worlds, fewer relational links, or there is some world for which the interpretation function is a subset of the interpretation function of M) or equal to M.

Assume that the frame of M' is smaller than the frame of M'. This implies M is not minimal because either M subset-simulates M' (when for some $u \in W'$ we have that $V'(u) \subset V(h(u))$) or M simulates M' (when for all $u \in W'$ we have that $V'(u) = V(h(u))$). The (subset-)simulation is simply the set $\{(u, h(u)) \mid u \in W'\}$. This contradicts the minimality of M. Hence, M' and M are based on the same frame.

Assume that for some $u \in W'$ we have that $V'(u) \subset V(h(u))$. This contradicts the assumption that M is a model minimal modulo subset-simulation because M' is subset-simulated by M (the subset-simulation is as in the previous case). This implies that for all $u \in W'$, $V'(u) = V(h(u))$.

As the frames and the interpretation functions of M and M' are the same, M and M' are the same model. This completes the proof. □

6 Minimal Model Soundness

Although the calculus is minimal model complete as presented up to now, it is not yet minimal model sound. This means, although among all the generated models there are all the minimal ones, the calculus does not generate only minimal models. However, as the calculus is minimal model complete, minimal model soundness can be achieved by closing all the branches of the tableau from which non-minimal models can be extracted. In order to prune properly the search space, we introduce a minimality test called *subset-simulation test*. This test allows us either to detect non-minimal models before they are completely computed, or to refine the minimal models found so far (that is, updating the set of minimal models by deleting models and inserting a new one).

Following an idea in [4,13], the aim of the minimal model test is to use previously extracted models to judge the minimality of the partial model that can be extracted from the currently selected branch. A crucial difference with [4,13] is that we cannot guarantee that the first extracted model is minimal. Our solution is to compute minimal models incrementally, meaning it is only known at the end of the complete derivation whether a model is minimal. The incremental generation of minimal models is achieved in the calculus by always selecting the left-most branch with the least number of worlds for further expansion first. This means, the calculus generates first all the models with the smallest domain, and then incrementally increases the domain size of the generated models. This expansion strategy alone is not enough to make the calculus minimal model sound because domain minimal models have good chances of not being minimal. Nevertheless, we think this is a good heuristic, because the minimality test can only be performed by comparing already extracted models with one (partial) model, and because the complexity of the algorithm presented in Section 4 depends on the number of worlds in the two models. Hence, finding domain minimal models first is likely to speed up the incremental generation of minimal models.

The *subset-simulation test* is divided into two cases. First, M is the partial model extracted from an open but not fully expanded branch. If there exists an already extracted model M' that is subset-simulated by M and M is not subset-simulated by M', then M is not minimal and the branch from which it was extracted is closed.

Second, M is the model extracted from an open and fully expanded branch. Then M is compared with the already extracted models and branches are closed accordingly. The closure of branches involves consideration of these three cases.

- M subset-simulates some minimal model M', but M' does not subset-simulate M. This means M is not minimal, and the branch from which M was extracted must be closed.
- M does not subset-simulate any minimal model M', but M' subset-simulates M. This means M' and all the models belonging to the symmetry class of M' are not minimal, and the branches from which those models were extracted must be closed.
- Some minimal model M' subset-simulates M, and M subset-simulates M'. This means M belongs to the same symmetry class of M'. Hence, simulation relationships between M and the models of the symmetry class need to be checked in order to refine the symmetry class. All the branches from models of the symmetry class which are no longer minimal must be closed.

The first case of the subset-simulation test allows us to prune the tree derivation before a branch is fully expanded. This is possible because if a partial model is already non-minimal, none of its possible extensions can be minimal. Hence, the branch can be closed without compromising minimal model completeness of the calculus.

As it is not always possible to recognise a non-minimal model before the branch is fully expanded, and because minimal models are computed incrementally by continuously refining the set of minimal models, the first case of the subset-simulation test is clearly not enough. The second case performs the refinement step of the current set of minimal models, meaning that even previously open and fully expanded branches can be closed. In other words, the second case requires checking subset-simulation relationships between M and representative models of all the symmetry classes of minimal models. This is because if M is subset-simulated by one model of a symmetry class, then it is subset-simulated by all of them due to subset-simulation being transitive.

From a theoretical perspective, it is not important when the minimality test is applied as long as it is always applied to open and full expanded branches (that is, as long as the second case of the minimality test is extensively performed). In order to avoid complex subset-simulation tests and to prune the derivation tree as soon as possible, heuristics can be used to fix the order of application of the rules and when the minimality test is performed. Our suggestion is to apply the rules in the following order: (SBR) rule, (α) rule, $(\Box)$ rule, (β) rule, and $(\Diamond)$ rule. The idea behind this order is to close a branch as soon as a contradiction occurs on the branch, and to delay the application of branching rules. Given this order of rule application, a sensible heuristic for the application of the minimality test is to perform it just before the application of the $(\Diamond)$ rule. This is because the $(\Diamond)$ rule has the highest branching factor, and the complexity of the subset-simulation test gradually increases after each application of this rule.

Using the proposed branch selection strategy and the minimality test, the calculus in Table 2 becomes minimal model sound.

Theorem 4. *Augmenting the tableau calculus with the subset-simulation test provides an approach that is minimal model sound when a fair expansion strategy is used. That is, it generates only models minimal modulo subset-simulation.*

7 Discussion

In the minimal model soundness theorem we required the expansion strategy of the calculus to be fair. The proposed expansion strategy to select the left-most branch with the least number of worlds can be seen as a variation of the common depth-first iterative deepening expansion strategy, where the weight used to select a branch is the number of worlds appearing on the branch. This strategy is not the only possible fair expansion strategy that can be applied to the calculus. Other variations of the depth-first iterative deepening strategy or a breadth-first strategy can also be used, and the resulting procedure is still minimal model sound and complete. Among the common strategies, depth-first

Table 3. Structural rules for extending the calculus. Note: all worlds in the conclusion of a rule with empty premises must already appear on the branch.

$$(\mathbf{T})\ \frac{}{(u,u) : R_i} \qquad\qquad (\mathbf{B})\ \frac{(u,v) : R_i}{(v,u) : R_i}$$

$$(\mathbf{4})\ \frac{(u,v) : R_i \quad (v,w) : R_i}{(u,w) : R_i} \qquad\qquad (\mathbf{5})\ \frac{(u,v) : R_i \quad (u,w) : R_i}{(v,w) : R_i}$$

$$(\mathbf{D})\ \frac{}{(u,u_1) : R_i \mid \ldots \mid (u,u_n) : R_i \mid (u,v) : R_i}$$

where u does not have an R_i-successor, each u_i appears on the branch, and v is fresh.

$$(\langle\mathcal{U}\rangle)\ \frac{u : \langle\mathcal{U}\rangle\phi}{u_1 : \phi \mid \ldots \mid u_n : \phi \mid v : \phi}$$

where each u_i appears on the branch, and v is fresh.

$$([\mathcal{U}])\ \frac{u : [\mathcal{U}]\phi}{v : \phi}$$

where v appears on the branch.

expansion is probably the only one that cannot be applied. This is because it is possible to have infinitely long branches, and depth-first expansion would not result in a complete tree derivation. As we have already pointed out, this is not the case for the multi-modal logic $\mathbf{K}_{(m)}$.

Even though the idea for the subset-simulation test is inspired by the model constraint propagation rule in [13,4], there are differences to that minimality test. The main difference is that we need the complete tree derivation for establishing which models are minimal, while in [13,4] this is not the case; the reason being that [13,4] are concerned with the generation of minimal Herbrand models, which means a subset (the set of minimal models) of a subset (the set of Herbrand models) of all possible models. Minimality modulo subset-simulation, instead, needs to evaluate many more models. It is interesting to note that if minimality modulo subset-simulation is applied only to Herbrand models, then the resulting set of minimal models is a refinement of the set of minimal Herbrand models.

The calculus in Table 2 can be extended easily to cover extensions of modal logic $\mathbf{K}_{(m)}$ by introducing rules that properly deal with such extensions. Table 3 shows the rules that allow the expansion of the tableau calculus to modal logics enriched with universal modalities, or to extensions in which the accessibility relations satisfy frame conditions from Table 1. Any extended version of the calculus results in a minimal model sound and complete tableau calculus as long as the minimality test and the described expansion strategy are used. A property that could be lost is termination of the calculus. We have already pointed out that the $(\Diamond)$ rule does not guarantee termination for the purpose of subset-simulation minimal model generation.

The extensions allowed by the rules in Table 3 are not the only possible extensions. One of the advantages of using minimality modulo subset-simulation is that the minimality criterion is applied to the graph representation of models. This means that the minimality criterion can be applied to all non-classical logics defined by a Kripke semantics. This includes logics such as modal logics, description logics, and temporal logics (even those that are not translatable to fragments of first-order logic). It is known from the literature, for example [6], that bisimulation needs to be extended depending on the expressivity of the logic. This is because bisimulation, like simulation and subset-simulation, is a local definition. It is however not required to extend the notion of subset-simulation for minimality modulo subset-simulation because the criterion requires full subset-simulation, changing the scope of the definition from local to global.

From the point of view of implementation, the calculus presents several challenges. Many well-known optimisation techniques such as backjumping or unit propagation, or a variation of them can be applied to speed up the implementation. The main problem, depending on the logic under consideration, is the possibility that the computation does not terminate. In this case, it might be sensible to impose a termination strategy at the cost of losing minimal model soundness and completeness, but preserving at least refutational soundness and completeness. This means not to stop the computation before the first model is found. After the first model has been found an early termination strategy can be used. This produces the best minimal models computed so far. As we are able to establish minimality of a model only in the complete derivation tree, stopping the computation at an early point does not guarantee the minimality of the models obtained so far. The idea of stopping the computation at a certain point can be seen as a branch and bound strategy, that is, the returned minimal models are the best minimal models extracted from the tableau up to this point. When to stop the derivation requires a new heuristic in the implementation, which one would probably make dependant on the domain of application. An alternative might be the use of a blocking mechanism such that the resulting procedure is strongly terminating. We are currently investigating blocking techniques to achieve strong termination while preserving minimal model soundness and completeness for logics with the finite model property. An appropriate blocking technique or a simplification of the ($\Diamond$) rule might result in a more efficient tableau calculus.

8 Conclusion

We presented minimality modulo subset-simulation as a novel minimality criterion for modal logics. The minimal models obtained following this new minimality criterion have the benefit that they reflect the semantics of a modal formula in a more faithful way than other minimality criteria. Although we emphasised the application of the criterion to the multi-modal logic $\mathbf{K}_{(m)}$, its semantic nature makes it applicable to a large number of non-classical logics.

We presented a minimal model complete tableau calculus for the multi-modal logic $\mathbf{K}_{(m)}$, and discussed how to achieve minimal model soundness through the use of the subset-simulation test. The resulting minimal model sound and complete calculus can easily be expanded to cover extensions of the multi-modal logic $\mathbf{K}_{(m)}$.

Even though the expansion rule for diamond formulae is expensive and termination is not always guaranteed, we believe that variations of the calculus can be efficiently implemented in such a way that the generated models are semantically meaningful and useful for applications. An implementation of the calculus, its extensions and variations can give important further insight regarding the generation of minimal models for non-classical logics.

References

1. Baader, F.: Least common subsumers and most specific concepts in a description logic with existential restrictions and terminological cycles. In: IJCAI 2003, pp. 319–324. Morgan Kaufmann (2003)
2. Baumgartner, P., Fürbach, 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)
3. Bry, F., Torge, S.: A deduction method complete for refutation and finite satisfiability. In: Dix, J., Fariñas del Cerro, L., Furbach, U. (eds.) JELIA 1998. LNCS (LNAI), vol. 1489, pp. 122–138. Springer, Heidelberg (1998)
4. Bry, F., Yahya, A.: Positive unit hyperresolution tableaux and their application to minimal model generation. J. Automated Reasoning 25(1), 35–82 (2000)
5. Clarke, E.M., Schlingloff, B.: Model checking. In: Handbook of Automated Reasoning, pp. 1635–1790. Elsevier (2001)
6. Divroodi, A.R., Nguyen, L.A.: On bisimulations for description logics. CoRR abs/1104.1964 (2011)
7. Gentilini, R., Piazza, C., Policriti, A.: From bisimulation to simulation - coarsest partition problems. J. Automated Reasoning 31, 73–103 (2002)
8. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proc. FCS-36, pp. 453–462. IEEE Computer Society (1995)
9. Hintikka, J.: Model minimization - an alternative to circumscription. J. Automated Reasoning 4(1), 1–13 (1988)
10. Lorenz, S.: A tableaux prover for domain minimization. J. Automated Reasoning 13(3), 375–390 (1994)
11. Lutz, C., Wolter, F.: Deciding inseparability and conservative extensions in the description logic $\mathcal{EL}$. J. Symbolic Computation 45(2), 194–228 (2010)
12. Niemelä, I.: Implementing circumscription using a tableau method. In: Proc. ECAI 1996, pp. 80–84. Wiley (1996)
13. Papacchini, F., Schmidt, R.A.: A tableau calculus for minimal modal model generation. ENTCS 278(3), 159–172 (2011)
14. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32(1), 57–95 (1987)

Hybrid Unification in the Description Logic $\mathcal{EL}$

Franz Baader, Oliver Fernández Gil, and Barbara Morawska[*]

Theoretical Computer Science, TU Dresden, Germany
{baader,morawska}@tcs.inf.tu-dresden.de,
fernandez@informatik.uni-leipzig.de

Abstract. Unification in Description Logics (DLs) has been proposed as an inference service that can, for example, be used to detect redundancies in ontologies. For the DL $\mathcal{EL}$, which is used to define several large biomedical ontologies, unification is NP-complete. However, the unification algorithms for $\mathcal{EL}$ developed until recently could not deal with ontologies containing general concept inclusions (GCIs). In a series of recent papers we have made some progress towards addressing this problem, but the ontologies the developed unification algorithms can deal with need to satisfy a certain cycle restriction. In the present paper, we follow a different approach. Instead of restricting the input ontologies, we generalize the notion of unifiers to so-called hybrid unifiers. Whereas classical unifiers can be viewed as acyclic TBoxes, hybrid unifiers are cyclic TBoxes, which are interpreted together with the ontology of the input using a hybrid semantics that combines fixpoint and descriptive semantics. We show that hybrid unification in $\mathcal{EL}$ is NP-complete and introduce a goal-oriented algorithm for computing hybrid unifiers.

1 Introduction

Description logics [5] are a well-investigated family of logic-based knowledge representation formalisms. They can be used to represent the relevant concepts of an application domain using concept descriptions, which are built from concept names and role names using certain concept constructors. The DL $\mathcal{EL}$, which offers the constructors conjunction ($\sqcap$), existential restriction ($\exists r.C$), and the top concept ($\top$), has recently drawn considerable attention since, on the one hand, important inference problems such as the subsumption problem are polynomial in $\mathcal{EL}$, even in the presence of GCIs [10]. On the other hand, though quite inexpressive, $\mathcal{EL}$ can be used to define biomedical ontologies, such as the large medical ontology SNOMED CT.[1] From a semantic point of view, concept names and concept descriptions represent sets of individuals, whereas role names represent binary relations between individuals. For example, using the concept names Head_injury and Severe, and the role names finding and status, we can describe the concept of a *patient with severe head injury* as

$$\mathsf{Patient} \sqcap \exists \mathsf{finding}.(\mathsf{Head\_injury} \sqcap \exists \mathsf{status}.\mathsf{Severe}). \tag{1}$$

[*] Supported by DFG under grant BA 1122/14-2.

[1] See http://www.ihtsdo.org/snomed-ct/

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 295–310, 2013.

In a DL ontology, one can use *concept definitions* to introduce abbreviations for concept descriptions. For example, we could use the definition $\mathsf{Head\_injury} \equiv \mathsf{Injury} \sqcap \exists \mathsf{finding\_site}.\mathsf{Head}$ to define $\mathsf{Head\_injury}$ as an injury that is located at the head. More generally, GCIs can be used to require that certain inclusions hold in all models of the ontology. For example,

$$\exists \mathsf{finding}.\exists \mathsf{status}.\mathsf{Severe} \sqsubseteq \exists \mathsf{status}.\mathsf{Emergency} \tag{2}$$

is a GCI that says that a severe finding entails an emergency status.

Knowledge representation systems based on DLs provide their users with various inference services that allow them to deduce implicit knowledge from the explicitly represented knowledge. For instance, the subsumption algorithm allows one to determine subconcept-superconcept relationships. For example, the concept description (1) is subsumed by (i.e., is a subconcept of) the concept description $\exists \mathsf{finding}.\exists \mathsf{status}.\mathsf{Severe}$. With respect to the GCI (2), it is thus also subsumed by $\exists \mathsf{status}.\mathsf{Emergency}$, i.e., in all models of this GCI, patients with severe head injury have an emergency status.

Unification in DLs has been proposed in [9] as a novel inference service that can, for instance, be used to detect redundancies in ontologies. For example, assume that one developer of a medical ontology describes the concept of a *patient with severe head injury* using the concept description (1), whereas another one represents it as

$$\mathsf{Patient} \sqcap \exists \mathsf{finding}.(\mathsf{Severe\_injury} \sqcap \exists \mathsf{finding\_site}.\mathsf{Head}). \tag{3}$$

These two concept descriptions are not equivalent, but they are nevertheless meant to represent the same concept. They can obviously be made equivalent by introducing definitions for the concept names $\mathsf{Head\_injury}$ and $\mathsf{Severe\_injury}$: if we define $\mathsf{Head\_injury} \equiv \mathsf{Injury} \sqcap \exists \mathsf{finding\_site}.\mathsf{Head}$ and $\mathsf{Severe\_injury} \equiv \mathsf{Injury} \sqcap \exists \mathsf{status}.\mathsf{Severe}$, then the two concept descriptions (1) and (3) are equivalent w.r.t. these definitions. If such definitions exist, we say that the descriptions are unifiable, and call the TBox consisting of these definitions a *unifier*. More precisely, it is required that this TBox is acyclic, i.e., there are no cyclic dependencies between the definitions.

To motivate our interest in unification w.r.t. GCIs, assume that the second developer uses the description

$$\mathsf{Patient} \sqcap \exists \mathsf{status}.\mathsf{Emergency} \sqcap \exists \mathsf{finding}.(\mathsf{Severe\_injury} \sqcap \exists \mathsf{finding\_site}.\mathsf{Head}) \tag{4}$$

instead of (3). The descriptions (1) and (4) are not unifiable without additional GCIs, but they are unifiable, with the same unifier as above, if the GCI (2) is present in a background ontology.

In [7], we were able to show that unification in the DL $\mathcal{EL}$ (without background ontology) is NP-complete. In addition to a brute-force "guess and then test" NP-algorithm [7], we have also developed a goal-oriented unification algorithm for $\mathcal{EL}$, in which nondeterministic decisions are only made if they are triggered by "unsolved parts" of the unification problem [8]. In [8] it was also shown that these two approaches for unification of $\mathcal{EL}$-concept descriptions (without any background ontology) can easily be extended to the case of an acyclic TBox as background ontology without really changing the algorithms or increasing their complexity. For more general GCIs, such a simple solution is no longer possible.

In [3], we extended the brute-force "guess and then test" NP-algorithm from [7] to the case of GCIs. Unfortunately, the algorithm is complete only for ontologies that satisfy a certain restriction on cycles, which, however, does not prevent all cycles. For example, the cyclic GCI $\exists\mathsf{child}.\mathsf{Human} \sqsubseteq \mathsf{Human}$ satisfies this restriction, whereas the cyclic GCI $\mathsf{Human} \sqsubseteq \exists\mathsf{parent}.\mathsf{Human}$ does not. In [4], we introduced a more practical, goal-oriented unification algorithm that can also deal with role hierarchies and transitive roles, but still needs the ontology (now consisting of GCIs and role axioms) to be cycle-restricted. At the moment, it is not clear how similar brute-force or goal-oriented algorithms could be obtained for the general case without cycle-restriction.

In this paper, we follow another line of attack on this problem. Instead of restricting the input ontology, we allow cyclic TBoxes to be used as unifiers. Subsumption w.r.t. cyclic TBoxes in $\mathcal{EL}$ has been investigated in detail in [1]. In addition to the classical descriptive semantics, it also makes sense to use *greatest fixpoint semantics (gfp-semantics)* for such TBoxes. For example, w.r.t. this semantics, the definition $X \equiv \exists\mathsf{parent}.X$ describes exactly those domain elements that are the origin of an infinite parent-chain, whereas descriptive semantics would also allow the empty set to be an interpretation of X, even if there are infinite parent-chains. *Hybrid semantics* deals with the case where a TBox interpreted with gfp-semantics is combined with GCIs that are interpreted with descriptive semantics [11,14,13]. Its introduction was originally motivated by the fact that the least common subsumer (lcs) w.r.t. a set of GCIs interpreted with descriptive semantics need not exist. For example, w.r.t. the GCIs

$$\mathsf{Human} \sqsubseteq \exists\mathsf{parent}.\mathsf{Human} \text{ and } \mathsf{Horse} \sqsubseteq \exists\mathsf{parent}.\mathsf{Horse}, \tag{5}$$

there is no least concept description (w.r.t. subsumption) that subsumes both Human and Horse. What elements of these two concepts have in common is that they are the origin of an infinite parent-chain, and thus the concept X with definition $X \equiv \exists\mathsf{parent}.X$ is their lcs, if we interpret this definition with gfp-semantics, but the GCIs (5) still with descriptive semantics. A hybrid unifier is a cyclic TBox that, together with the background ontology consisting of GCIs, entails the unification problem w.r.t. hybrid semantics. We will show that hybrid unification in $\mathcal{EL}$, i.e., the problem of testing whether a hybrid unifier exists, is NP-complete. In addition, we will introduce a goal-oriented algorithm for computing hybrid unifiers. The proofs, which can be found in [6], are based on the proof system for hybrid subsumption introduced in [14,13].

2 The Description Logic $\mathcal{EL}$

The expressiveness of a DL is determined both by the formalism for describing concepts (the concept description language) and the terminological formalism, which can be used to state additional constraints on the interpretation of concepts in a so-called ontology.

The Concept Description Language. The *concept description language* considered in this paper is called $\mathcal{EL}$. Starting with a finite set N_C of *concept names* and a finite set N_R of *role names*, $\mathcal{EL}$*-concept descriptions* are built from concept names using the constructors *conjunction* ($C \sqcap D$), *existential restriction* ($\exists r.C$ for every $r \in N_R$), and *top* ($\top$). Since in this paper we only consider $\mathcal{EL}$-concept descriptions, we will usually dispense with the prefix $\mathcal{EL}$.

On the *semantic side*, concept descriptions are interpreted as sets. To be more precise, an *interpretation* $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ consists of a non-empty domain $\Delta^\mathcal{I}$ and an interpretation function $\cdot^\mathcal{I}$ that maps concept names to subsets of $\Delta^\mathcal{I}$ and role names to binary relations over $\Delta^\mathcal{I}$. This function is inductively extended to concept descriptions as follows:

$$\top^\mathcal{I} := \Delta^\mathcal{I}, \ \ (C \sqcap D)^\mathcal{I} := C^\mathcal{I} \cap D^\mathcal{I}, \ \ (\exists r.C)^\mathcal{I} := \{x \mid \exists y : (x,y) \in r^\mathcal{I} \wedge y \in C^\mathcal{I}\}$$

Classical Ontologies and Subsumption. A *concept definition* is an expression of the form $X \equiv C$ where X is a concept name and C is a concept description, and a *general concept inclusion* (GCI) is an expression of the form $C \sqsubseteq D$, where C, D are concept descriptions. An interpretation $\mathcal{I}$ is a *model* of this concept definition (this GCI) if it satisfies $X^\mathcal{I} = C^\mathcal{I}$ ($C^\mathcal{I} \subseteq D^\mathcal{I}$). This semantics for GCIs and concept definitions is usually called *descriptive semantics.*

A *TBox* is a finite set $\mathcal{T}$ of concept definitions that does not contain multiple definitions, i.e., $\{X \equiv C, X \equiv D\} \subseteq \mathcal{T}$ implies $C = D$. Note that we do *not* prohibit cyclic dependencies among the concept definitions in a TBox, i.e., when defining a concept X we may (directly or indirectly) refer to X. An *acyclic TBox* is a TBox without cyclic dependencies. An *ontology* is a finite set of GCIs. The interpretation $\mathcal{I}$ is a *model* of a TBox (ontology) iff it is a model of all concept definitions (GCIs) contained in it.

A concept description C is *subsumed* by a concept description D w.r.t. an ontology $\mathcal{O}$ (written $C \sqsubseteq_\mathcal{O} D$) if every model of $\mathcal{O}$ is also a model of the GCI $C \sqsubseteq D$. We say that C is *equivalent* to D w.r.t. $\mathcal{O}$ ($C \equiv_\mathcal{O} D$) if $C \sqsubseteq_\mathcal{O} D$ and $D \sqsubseteq_\mathcal{O} C$. As shown in [10], subsumption w.r.t. $\mathcal{EL}$-ontologies is decidable in polynomial time.

Note that TBoxes can be seen as special kinds of ontologies since concept definitions $X \equiv C$ can of course be expressed by GCIs $X \sqsubseteq C, C \sqsubseteq X$. Thus, the above definition of subsumption also applies to TBoxes. However, in our hybrid ontologies we will interpret concept definitions using greatest fixpoint semantics rather than descriptive semantics.

Hybrid Ontologies. We assume in the following that the set of concept names N_C is partitioned into the set of *primitive concepts* N_{prim} and the set of *defined concepts* N_{def}. In a hybrid TBox, concept names occurring on the left-hand side of a concept definition are required to come from the set N_{def}, whereas GCIs must not contain concept names from N_{def}.

Definition 1 (Hybrid $\mathcal{EL}$-ontologies). *A* hybrid $\mathcal{EL}$-ontology *is a pair* $(\mathcal{O}, \mathcal{T})$*, where* $\mathcal{O}$ *is an* $\mathcal{EL}$*-ontology containing only concept names from* N_{prim}*, and* $\mathcal{T}$ *is a (possibly cyclic)* $\mathcal{EL}$*-TBox such that* $X \equiv C \in \mathcal{T}$ *for some concept description* C *iff* $X \in N_{def}$*.*

The idea underlying the definition of hybrid ontologies is the following: $\mathcal{O}$ can be used to constrain the interpretation of the primitive concepts and roles, whereas $\mathcal{T}$ tells us how to interpret the defined concepts occurring in it, once the interpretation of the primitive concepts and roles is fixed.

A *primitive interpretation* $\mathcal{J}$ is defined like an interpretation, with the only difference that it does not provide an interpretation for the defined concepts. A primitive interpretation can thus interpret concept descriptions built over N_{prim} and N_R, but it cannot interpret concept descriptions containing elements of N_{def}. Given a primitive interpretation $\mathcal{J}$, we say that the (full) interpretation $\mathcal{I}$ is *based on* $\mathcal{J}$ if it has the same domain as $\mathcal{J}$ and its interpretation function coincides with $\mathcal{J}$ on N_{prim} and N_R.

Given two interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$ based on the same primitive interpretation $\mathcal{J}$, we define $\mathcal{I}_1 \preceq_{\mathcal{J}} \mathcal{I}_2$ iff $X^{\mathcal{I}_1} \subseteq X^{\mathcal{I}_2}$ for all $X \in N_{def}$.

It is easy to see that the relation $\preceq_{\mathcal{J}}$ is a partial order on the set of interpretations based on $\mathcal{J}$. In [1] the following was shown: given an $\mathcal{EL}$-TBox $\mathcal{T}$ and a primitive interpretation $\mathcal{J}$, there exists a unique model $\mathcal{I}$ of $\mathcal{T}$ such that

- $\mathcal{I}$ is based on $\mathcal{J}$;
- $\mathcal{I}' \preceq_{\mathcal{J}} \mathcal{I}$ for all models $\mathcal{I}'$ of $\mathcal{T}$ that are based on $\mathcal{J}$.

We call such a model $\mathcal{I}$ a *gfp-model* of $\mathcal{T}$.

Definition 2 (Semantics of hybrid $\mathcal{EL}$-ontologies). *The interpretation* $\mathcal{I}$ *is a* hybrid model *of the hybrid* $\mathcal{EL}$*-ontology* $(\mathcal{O}, \mathcal{T})$ *iff* $\mathcal{I}$ *is a gfp-model of* $\mathcal{T}$ *and the primitive interpretation* $\mathcal{J}$ *it is based on is a model of* $\mathcal{O}$*.*

It is well-known that gfp-semantics coincides with descriptive semantics for acyclic TBoxes. Thus, if $\mathcal{T}$ is actually acyclic, then $\mathcal{I}$ is a hybrid model of $(\mathcal{O}, \mathcal{T})$ according to the semantics introduced in Definition 2 iff it is a model of $\mathcal{T} \cup \mathcal{O}$ w.r.t. descriptive semantics, i.e., iff $\mathcal{I}$ is a model of every GCI in $\mathcal{O}$ and of every concept definition in $\mathcal{T}$.

Subsumption w.r.t. Hybrid $\mathcal{EL}$-Ontologies. Let $(\mathcal{O}, \mathcal{T})$ be a hybrid $\mathcal{EL}$-ontology and C, D $\mathcal{EL}$-concept descriptions. Then C *is subsumed by* D *w.r.t.* $(\mathcal{O}, \mathcal{T})$ (written $C \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} D$) iff every hybrid model of $(\mathcal{O}, \mathcal{T})$ is also a model of the GCI $C \sqsubseteq D$. As shown in [11,14,13], subsumption w.r.t. hybrid $\mathcal{EL}$-ontologies is also decidable in polynomial time.

Here, we sketch the proof-theoretic approach for deciding subsumption from [14,13] since our algorithms for hybrid unification in $\mathcal{EL}$ are based on it. The proof calculus is parametrized with a hybrid $\mathcal{EL}$-ontology $(\mathcal{O}, \mathcal{T})$ and a finite set of GCIs Δ for which we want to decide subsumption. A *sequent for* $(\mathcal{O}, \mathcal{T})$ *and* Δ is of the form $C \sqsubseteq_n D$, where C, D are sub-descriptions of concept descriptions

$$C \sqsubseteq_n C \quad \text{(Refl)} \qquad C \sqsubseteq_n \top \quad \text{(Top)} \qquad C \sqsubseteq_0 D \quad \text{(Start)}$$

$$\frac{C \sqsubseteq_n E}{C \sqcap D \sqsubseteq_n E} \ \text{(AndL1)} \qquad \frac{D \sqsubseteq_n E}{C \sqcap D \sqsubseteq_n E} \ \text{(AndL2)} \qquad \frac{C \sqsubseteq_n D \quad C \sqsubseteq_n E}{C \sqsubseteq_n D \sqcap E} \ \text{(AndR)}$$

$$\frac{C \sqsubseteq_n D}{\exists r.C \sqsubseteq_n \exists r.D} \ \text{(Ex)}$$

$$\frac{C \sqsubseteq_n D}{X \sqsubseteq_n D} \ \text{(DefL)} \qquad \frac{D \sqsubseteq_n C}{D \sqsubseteq_{n+1} X} \ \text{(DefR)} \qquad \frac{C \sqsubseteq_n E \quad F \sqsubseteq_n D}{C \sqsubseteq_n D} \ \text{(GCI)}$$

$$\text{for } X \equiv C \in \mathcal{T} \qquad \text{for } X \equiv C \in \mathcal{T} \qquad \text{for } E \sqsubseteq F \in \mathcal{O}$$

Fig. 1. The calculus HC($\mathcal{O}, \mathcal{T}, \Delta$)

occurring in $\mathcal{O}$, $\mathcal{T}$, and Δ, and $n \geq 0$. If $(\mathcal{O}, \mathcal{T})$ and Δ are clear from the context, we will sometimes simply say sequent without specifying $(\mathcal{O}, \mathcal{T})$ and Δ explicitly.

The rules of the **H**ybrid $\mathcal{EL}$-ontology **C**alculus HC($\mathcal{O}, \mathcal{T}, \Delta$) are depicted in Fig. 1. Again, if $(\mathcal{O}, \mathcal{T})$ and Δ are clear from the context, we will sometimes dispense with specifying them explicitly and just talk about the calculus HC. The rules of this calculus can be used to derive new sequents from sequents that have already been derived. For example, the sequents in the first row of the figure can always be derived without any prerequisites, using the rules (Refl), (Top), and (Start), respectively. Using the rule (AndR), the sequent $C \sqsubseteq_n D \sqcap E$ can be derived in case both $C \sqsubseteq_n D$ and $C \sqsubseteq_n E$ have already been derived. Note that the rule Start applies only for $n = 0$. Also note that, in the rule (DefR), the index is incremented when going from the prerequisite to the consequent.

A derivation in HC($\mathcal{O}, \mathcal{T}, \Delta$) can be represented in an obvious way by a proof tree whose nodes are sequents: a proof tree for $C \sqsubseteq_n D$ has this sequent as its root, instances of the rules Refl, Top, and Start as leaves, and each parent-child relation corresponds to an instance of a rule of HC other than Refl, Top, and Start (see [14,13] for more details)

Definition 3. *Let C, D be sub-descriptions of concept descriptions occurring in $\mathcal{O}, \mathcal{T}$, and Δ. Then we say that $C \sqsubseteq_\infty D$* can be derived *in* HC($\mathcal{O}, \mathcal{T}, \Delta$) *if all sequents $C \sqsubseteq_n D$ for $n \geq 0$ can be derived using the rules of* HC($\mathcal{O}, \mathcal{T}, \Delta$).

The calculus HC is sound and complete for subsumption w.r.t. hybrid $\mathcal{EL}$-ontologies in the following sense.

Theorem 4 (Soundness and Completeness of HC). *Let $(\mathcal{O}, \mathcal{T})$ be a hybrid $\mathcal{EL}$-TBox, Δ a finite set of GCIs, and C, D sub-descriptions of concept descriptions occurring in $\mathcal{O}, \mathcal{T}$, and Δ. Then $C \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} D$ iff $C \sqsubseteq_\infty D$ can be derived in* HC($\mathcal{O}, \mathcal{T}, \Delta$).

In [13], soundness and completeness of HC is actually formulated for a restricted setting where Δ is empty and C, D are elements of N_{def} that occur as left-hand sides in $\mathcal{T}$. It is, however, easy to see that the proof given in [13] generalizes to the above theorem.

For $n \in \mathbb{N} \cup \{\infty\}$, we collect the GCIs $C \sqsubseteq D$ such that $C \sqsubseteq_n D$ is derivable in HC$(\mathcal{O}, \mathcal{T}, \Delta)$ in the set $\mathcal{D}_n(\mathcal{O}, \mathcal{T}, \Delta)$. Obviously, $\mathcal{D}_0(\mathcal{O}, \mathcal{T}, \Delta)$ consists of all GCIs built from sub-descriptions of concept descriptions occurring in $\mathcal{O}, \mathcal{T}$, and Δ, and it is not hard to show that $\mathcal{D}_{n+1}(\mathcal{O}, \mathcal{T}, \Delta) \subseteq \mathcal{D}_n(\mathcal{O}, \mathcal{T}, \Delta)$ holds for all $n \geq 0$ [14,13]. Thus, to compute $\mathcal{D}_\infty(\mathcal{O}, \mathcal{T}, \Delta)$, one can start with $\mathcal{D}_0(\mathcal{O}, \mathcal{T}, \Delta)$, and then compute $\mathcal{D}_1(\mathcal{O}, \mathcal{T}, \Delta), \mathcal{D}_2(\mathcal{O}, \mathcal{T}, \Delta), \ldots$, until $\mathcal{D}_{m+1}(\mathcal{O}, \mathcal{T}, \Delta) = \mathcal{D}_m(\mathcal{O}, \mathcal{T}, \Delta)$ holds for some $m \geq 0$, and thus $\mathcal{D}_m(\mathcal{O}, \mathcal{T}, \Delta) = \mathcal{D}_\infty(\mathcal{O}, \mathcal{T}, \Delta)$. Since the cardinality of the set of sub-descriptions is polynomial in the size of the input $\mathcal{O}, \mathcal{T}$, and Δ, the computation of each set $\mathcal{D}_n(\mathcal{O}, \mathcal{T}, \Delta)$ can be done in polynomial time, and we can be sure that only polynomially many such sets need to be computed until an m with $\mathcal{D}_{m+1}(\mathcal{O}, \mathcal{T}, \Delta) = \mathcal{D}_m(\mathcal{O}, \mathcal{T}, \Delta)$ is reached. This shows that the calculus HC$(\mathcal{O}, \mathcal{T}, \Delta)$ indeed yields a polynomial-time subsumption algorithm (see [14,13] for details).

3 Hybrid Unification in $\mathcal{EL}$

We will first introduce the new notion of hybrid unification and then relate it to the notion of unification in $\mathcal{EL}$ w.r.t. background ontologies considered in [3,4].

Definition 5. *Let $\mathcal{O}$ be an $\mathcal{EL}$-ontology containing only concept names from N_{prim}. An $\mathcal{EL}$-*unification problem *w.r.t. $\mathcal{O}$ is a finite set of GCIs $\Gamma = \{C_1 \sqsubseteq D_1, \ldots, C_n \sqsubseteq D_n\}$ (which may also contain concept names from N_{def}). The TBox $\mathcal{T}$ is a* hybrid unifier *of Γ w.r.t. $\mathcal{O}$ if $(\mathcal{O}, \mathcal{T})$ is a hybrid $\mathcal{EL}$-ontology that entails all the GCIs in Γ, i.e. , $C_1 \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} D_1, \ldots, C_n \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} D_n$. We call such a TBox $\mathcal{T}$ a* classical unifier *of Γ w.r.t. $\mathcal{O}$ if it is acyclic.*

It is easy to see that the notion of a classical unifier indeed corresponds to the notion of a unifier introduced in [3,4]. In fact, N_{prim} and N_{def} respectively correspond to the sets of concept constants and concept variables in previous papers on unification in DLs. Using acyclic TBoxes rather than substitutions as unifiers is also not a relevant difference. As explained in [2], by unfolding concept definitions, the acyclic TBox $\mathcal{T}$ can be transformed into a substitution $\sigma_{\mathcal{T}}$ such that $C_i \sqsubseteq_{\mathcal{T} \cup \mathcal{O}} D_i$ iff $\sigma_{\mathcal{T}}(C_i) \sqsubseteq_{\mathcal{O}} \sigma_{\mathcal{T}}(D_i)$. Conversely, replacements $X \mapsto E$ of a substitution σ can be expressed as concept definitions $X \equiv E$ in a corresponding acyclic TBox. In contrast, hybrid unifiers cannot be translated into substitutions since the unfolding process would not terminate for a cyclic TBox.

Obviously, any classical unifier is a hybrid unifier, but the converse need not hold. The following is an example of an $\mathcal{EL}$-unification problem w.r.t. a background ontology that has a hybrid unifier, but no classical unifier.

Example 6. Let $\mathcal{O}$ be the ontology consisting of the GCIs (5), and

$$\Gamma := \{\mathsf{Human} \sqsubseteq X, \mathsf{Horse} \sqsubseteq X, X \sqsubseteq \exists \mathsf{parent}.X\},$$

where $X \in N_{def}$ and $\mathsf{Human}, \mathsf{Horse} \in N_{prim}$. Intuitively, this unification problem asks for a concept such that all horses and humans belong to this concept and every element of it has a parent also belonging to it. It is easy to see that $\mathcal{T} := \{X \equiv \exists \mathsf{parent}.X\}$ is a hybrid unifier of Γ w.r.t. $\mathcal{O}$. In fact, we have already mentioned in the introduction that X is then the lcs of Human and Horse, and obviously the hybrid ontology $(\mathcal{O}, \mathcal{T})$ also entails the third GCI in Γ. It is also not hard to show that this unification problem does not have a classical unifier, basically for the same reasons that Human and Horse do not have an $\mathcal{EL}$-concept description as lcs (see [6] for details).

Flat Unification Problems. To simplify the technical development, it is convenient to normalize the unification problem appropriately. To introduce this normal form, we need the notion of an atom. An *atom* is a concept name or an existential restriction. Obviously, every $\mathcal{EL}$-concept description C is a finite conjunction of atoms, where $\top$ is considered to be the empty conjunction. An atom is called *flat* if it is a concept name or an existential restriction of the form $\exists r.A$ for a concept name A.

The GCI $C \sqsubseteq D$ is called *flat* if C is a conjunction of $n \geq 0$ flat atoms and D is a flat atom. The unification problem Γ w.r.t. the ontology $\mathcal{O}$ is called *flat* if both Γ and $\mathcal{O}$ consist of flat GCIs.

Given a unification problem Γ w.r.t. an ontology $\mathcal{O}$, we can compute in polynomial time (see [6]) a flat ontology $\mathcal{O}'$ and a flat unification problem Γ' such that Γ has a (hybrid or classical) unifier w.r.t. $\mathcal{O}$ iff Γ' has a (hybrid or classical) unifier w.r.t. $\mathcal{O}'$. For this reason, we will assume in the following that all unification problems are flat.

Local Unifiers. The main reason why $\mathcal{EL}$-unification without background ontologies is in NP is that any unification problem that has a unifier also has a local unifier. For classical unification w.r.t. background ontologies this is only true if the background ontology is cycle-restricted.

Given a flat unification problem Γ w.r.t. an ontology $\mathcal{O}$, we denote by At the set of atoms occurring as sub-descriptions in GCIs in Γ or $\mathcal{O}$. The set of *non-variable atoms* is defined by $\mathrm{At}_{\mathsf{nv}} := \mathrm{At} \setminus N_{def}$. Though the elements of $\mathrm{At}_{\mathsf{nv}}$ cannot be defined concepts, they may contain defined concepts if they are of the form $\exists r.X$ for some role r and a concept name $X \in N_{def}$.

In order to define local unifiers, we consider assignments ζ of subsets ζ_X of $\mathrm{At}_{\mathsf{nv}}$ to defined concepts $X \in N_{def}$. Such an assignment induces a TBox

$$T_\zeta := \{X \equiv \bigsqcap_{D \in \zeta_X} D \mid X \in N_{def}\}.$$

We call such a TBox *local*. The (hybrid or classical) unifier $\mathcal{T}$ of Γ w.r.t. $\mathcal{O}$ is called *local unifier* if $\mathcal{T}$ is local, i.e., there is an assignment ζ such that $\mathcal{T} = T_\zeta$.

As shown in [3], there are unification problems that have a classical unifier, but no local classical unifier.

Example 7. Let $\mathcal{O} = \{B \sqsubseteq \exists s.D,\ D \sqsubseteq B\}$ and consider the unification problem

$$\begin{aligned}\Gamma := \{&A_1 \sqcap B \sqsubseteq Y_1,\ Y_1 \sqsubseteq A_1 \sqcap B,\ A_2 \sqcap B \sqsubseteq Y_2, Y_2 \sqsubseteq A_2 \sqcap B,\\ &\exists s.Y_1 \sqsubseteq X,\ \exists s.Y_2 \sqsubseteq X,\ X \sqsubseteq \exists s.X\},\end{aligned}$$

where $A_1, A_2, B \in N_{prim}$ and $X, Y_1, Y_2 \in N_{def}$. This problem has the classical unifier $\mathcal{T} := \{Y_1 \equiv A_1 \sqcap B, Y_2 \equiv A_2 \sqcap B, X \equiv \exists s.B\}$, which is not local since it uses the atom $\exists s.B$. As shown in [3], Γ actually does not have a local classical unifier w.r.t. $\mathcal{O}$. However, it is easy to see that $\mathcal{T} := \{Y_1 \equiv A_1 \sqcap B, Y_2 \equiv A_2 \sqcap B, X \equiv \exists s.X\}$ is a local hybrid unifier of $\mathcal{T}$. In fact, gfp-semantics applied to $\mathcal{T}$ ensures that X consists of exactly those domain elements that are the origin of an infinite s-chain, and $\mathcal{O}$ ensures that any element of B (and thus also of $\exists s.B$) is the origin of an infinite s-chain.

To overcome the problem of missing local unifiers, the notion of a cycle-restricted ontology was introduced in [3]: the $\mathcal{EL}$-ontology $\mathcal{O}$ is called *cycle-restricted* if there is no nonempty sequence $r_1, \ldots, r_n$ of role names and $\mathcal{EL}$-concept description C such that $C \sqsubseteq_{\mathcal{O}} \exists r_1. \cdots \exists r_n.C$. Note that the ontology $\mathcal{O}$ of Example 7 is not cycle-restricted since $B \sqsubseteq_{\mathcal{O}} \exists s.B$.

The main technical result shown in [3] is that any $\mathcal{EL}$-unification problem Γ that has a classical unifier w.r.t. the cycle-restricted ontology $\mathcal{O}$ also has a local classical unifier. This yields the following brute-force algorithm for classical $\mathcal{EL}$-unification w.r.t. cycle-restricted ontologies: first guess an acyclic local TBox $\mathcal{T}$, and then check whether $\mathcal{T}$ is indeed a unifier of Γ w.r.t. $\mathcal{O}$. As shown in [3], this algorithm runs in nondeterministic polynomial time. NP-hardness follows from the fact that already classical unification in $\mathcal{EL}$ w.r.t. the empty ontology is NP-hard [7].

4 Hybrid $\mathcal{EL}$-Unification is NP-Complete

The fact that hybrid $\mathcal{EL}$-unification w.r.t. arbitrary $\mathcal{EL}$-ontologies is in NP is an easy consequence of the following proposition.

Proposition 8. *Consider a flat $\mathcal{EL}$-unification problem Γ w.r.t. an $\mathcal{EL}$-ontology $\mathcal{O}$. If Γ has a hybrid unifier w.r.t. $\mathcal{O}$ then it has a* local *hybrid unifier w.r.t. $\mathcal{O}$.*

In fact, the NP-algorithm simply guesses a local TBox and then checks (using the polynomial-time algorithm for hybrid subsumption) whether it is a hybrid unifier.

To prove the proposition, we assume that $\mathcal{T}$ is a hybrid unifier of Γ w.r.t. $\mathcal{O}$. We use this unifier to define an assignment $\zeta^{\mathcal{T}}$ as follows:

$$\zeta^{\mathcal{T}}_X := \{D \in \mathrm{At}_{\mathsf{nv}} \mid X \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} D\}.$$

Let $\mathcal{T}'$ be the TBox induced by this assignment. To show that $\mathcal{T}'$ is indeed a hybrid unifier of Γ w.r.t. $\mathcal{O}$, we consider the set of GCIs

$$\Delta := \{C_1 \sqcap \ldots \sqcap C_m \sqsubseteq D \mid C_1, \ldots, C_m, D \in \mathrm{At}\},$$

and prove that, for any GCI $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq D \in \Delta$, derivability of $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_\infty D$ in $\mathrm{HC}(\mathcal{O}, \mathcal{T}, \Delta)$ implies derivability of $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_\infty D$ also in $\mathrm{HC}(\mathcal{O}, \mathcal{T}', \Delta)$. Soundness and completeness of HC, together with the facts that $\Gamma \subseteq \Delta$ and $\mathcal{T}$ is a hybrid unifier of Γ w.r.t. $\mathcal{O}$, then imply that $\mathcal{T}'$ is also a hybrid unifier of Γ w.r.t. $\mathcal{O}$. Thus, to complete the proof of Proposition 8, it is enough to prove the following lemma.

Lemma 9. *Let $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq D \in \Delta$. If $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_\infty D$ is derivable in* $\mathrm{HC}(\mathcal{O}, \mathcal{T}, \Delta)$*, then $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_n D$ is derivable in* $\mathrm{HC}(\mathcal{O}, \mathcal{T}', \Delta)$ *for all $n \geq 0$.*

Proof. We prove derivability of $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_n D$ in $\mathrm{HC}(\mathcal{O}, \mathcal{T}', \Delta)$ by induction on n. The *base case* is trivial due to the rule (Start).

Induction Step: We assume that the statement of the lemma holds for $n-1$, and show that it then also holds for n. Let ℓ be such that $\mathcal{D}_\ell(\mathcal{O}, \mathcal{T}, \Delta) = \mathcal{D}_\infty(\mathcal{O}, \mathcal{T}, \Delta)$. We know that there exists a proof tree $\mathcal{P}$ for $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_\ell D$ in $\mathrm{HC}(\mathcal{O}, \mathcal{T}, \Delta)$. Consider the subtree of $\mathcal{P}$ that is obtained from it by cutting branches at the nodes obtained by an application of one of the rules (DefL) or (DefR). The tree obtained this way contains only sequents with index ℓ and has as its leaves

- instances of the rules (Refl), (Top), or (Start),
- consequences $E_1 \sqsubseteq_\ell E_2$ of instances of the rules (DefL) or (DefR).

In order to show that $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_n D$ is derivable in $\mathrm{HC}(\mathcal{O}, \mathcal{T}', \Delta)$, it is sufficient to show that, for leaves $E_1 \sqsubseteq_\ell E_2$ of the second kind, $E_1 \sqsubseteq_n E_2$ is derivable in $\mathrm{HC}(\mathcal{O}, \mathcal{T}', \Delta)$ (see [6] for details).

First, assume that $E_1 \sqsubseteq_\ell E_2$ was obtained by an application of (DefR). Then $E_2 \in N_{def}$. Assume that $\zeta^{\mathcal{T}}_{E_2} = \{F_1, \ldots, F_q\}$. By the definition of $\zeta^{\mathcal{T}}$, we have $E_2 \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} F_i$ for all $i, 1 \leq i \leq q$. In addition, by our choice of ℓ, derivability of $E_1 \sqsubseteq_\ell E_2$ in $\mathrm{HC}(\mathcal{O}, \mathcal{T}, \Delta)$ (using the subtree of $\mathcal{P}$ with this node as root) yields $E_1 \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} E_2$, and thus $E_1 \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} F_i$ for all $i, 1 \leq i \leq q$. Consequently, $E_1 \sqsubseteq_\infty F_i$ is derivable in $\mathrm{HC}(\mathcal{O}, \mathcal{T}, \Delta)$ for all $i, 1 \leq i \leq q$. Since E_1 is a conjunction of elements of At and $F_1, \ldots, F_q \in \mathrm{At}$, induction yields that $E_1 \sqsubseteq_{n-1} F_i$ is derivable in $\mathrm{HC}(\mathcal{O}, \mathcal{T}', \Delta)$ for all $i, 1 \leq i \leq q$. Performing $q-1$ applications of (AndR) thus allows us to derive $E_1 \sqsubseteq_{n-1} F_1 \sqcap \ldots \sqcap F_q$ in $\mathrm{HC}(\mathcal{O}, \mathcal{T}', \Delta)$. Since $\mathcal{T}'$ contains the definition $E_2 \equiv F_1 \sqcap \ldots \sqcap F_q$, an application of (DefR) shows that $E_1 \sqsubseteq_n E_2$ is derivable in $\mathrm{HC}(\mathcal{O}, \mathcal{T}', \Delta)$.

Second, assume that $E_1 \sqsubseteq_\ell E_2$ was obtained by an application of (DefL). Then $E_1 \in N_{def}$ and $E_2 = F_1 \sqcap \ldots \sqcap F_m$ for elements $F_1, \ldots, F_m$ of At. By our choice of ℓ we have $E_1 \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} E_2$, and thus $E_1 \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} F_i$ for all $i, 1 \leq i \leq q$. It is sufficient to show, for all $i, 1 \leq i \leq q$, that $E_1 \sqsubseteq_n F_i$ is derivable in $\mathrm{HC}(\mathcal{O}, \mathcal{T}', \Delta)$ since $q-1$ applications of (AndR) then yield derivability of $E_1 \sqsubseteq_n E_2$ in $\mathrm{HC}(\mathcal{O}, \mathcal{T}', \Delta)$.

If F_i does not belong to N_{def}, then it is an element of $\mathrm{At}_{\mathsf{nv}}$. The definition of $\zeta^{\mathcal{T}}$ thus yields $F_i \in \zeta^{\mathcal{T}}_{E_1}$. Consequently, F_i occurs as a conjunct on the right-hand side of the definition of E_1 in $\mathcal{T}'$. This implies $E_1 \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}'} F_i$, and thus $E_1 \sqsubseteq_n F_i$ is derivable in $\mathrm{HC}(\mathcal{O}, \mathcal{T}', \Delta)$.

If $F_i \in N_{def}$, then $E_1 \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} F_i$ implies that $\zeta^{\mathcal{T}}_{F_i} \subseteq \zeta^{\mathcal{T}}_{E_1}$. Consequently, every conjunct on the right-hand side of the definition of F_i in $\mathcal{T}'$ is also a conjunct on the right-hand side of the definition of E_1 in $\mathcal{T}'$. This implies $E_1 \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}'} F_i$, and thus $E_1 \sqsubseteq_n F_i$ is derivable in HC$(\mathcal{O}, \mathcal{T}', \Delta)$. □

This finishes the proof of Proposition 8, and thus shows that hybrid $\mathcal{EL}$-unification w.r.t. arbitrary $\mathcal{EL}$-ontologies is in NP. NP-hardness does *not* follow directly from NP-hardness of classical $\mathcal{EL}$-unification. In fact, as we have seen in Example 6, an $\mathcal{EL}$-unification problem that does not have a classical unifier may well have a hybrid unifier. Instead, we reduce $\mathcal{EL}$-matching modulo equivalence to hybrid $\mathcal{EL}$-unification.

Using the notions introduced in this paper, $\mathcal{EL}$-matching modulo equivalence can be defined as follows. An *$\mathcal{EL}$-matching problem modulo equivalence* is an $\mathcal{EL}$-unification problem of the form $\{C \sqsubseteq D, D \sqsubseteq C\}$ such that D does not contain elements of N_{def}. A *matcher* of such a problem is a classical unifier of it. As shown in [12], testing whether a matching problem modulo equivalence has a matcher or not is an NP-complete problem. Thus, NP-hardness of hybrid $\mathcal{EL}$-unification w.r.t. $\mathcal{EL}$-ontologies is an immediate consequence of the following lemma, whose (non-trivial) proof can be found in [6].

Lemma 10. *If an $\mathcal{EL}$-matching problem modulo equivalence has a hybrid unifier w.r.t. the empty ontology, then it also has a matcher.*

To sum up, we have thus determine the exact worst-case complexity of hybrid $\mathcal{EL}$-unification.

Theorem 11. *The problem of testing whether an $\mathcal{EL}$-unification problem w.r.t. an arbitrary $\mathcal{EL}$-ontology has a hybrid unifier or not is* NP*-complete.*

5 A Goal-Oriented Algorithm for Hybrid $\mathcal{EL}$-Unification

The brute-force algorithm is not practical since it blindly guesses a local TBox and only afterwards checks whether the guessed TBox is a hybrid unifier. We now introduce a more goal-oriented unification algorithm, in which nondeterministic decisions are only made if they are triggered by "unsolved parts" of the unification problem. In addition, failure due to wrong guesses can be detected early. Any non-failing run of the algorithm produces a hybrid unifier, i.e., there is no need for checking whether the TBox computed by this run really is a hybrid unifier. This goal-oriented algorithm is based on ideas similar to the ones used in the algorithm for classical unification in $\mathcal{EL}$ w.r.t. cycle-restricted ontologies in [4]. However, it differs from the previous algorithm in several respects.

First, it is based on the proof calculus HC rather than on a structural characterization of subsumption, as employed in [4]. Basically, to solve the unification problem Γ w.r.t. the ontology $\mathcal{O}$, the rules of the algorithm try to build, for each GCI $C \sqsubseteq D \in \Gamma$, a proof tree for the sequent $C \sqsubseteq_\ell D$ while simultaneously generating the hybrid unifier $\mathcal{T}$ by adding non-variable atoms to an assignment

ζ inducing $\mathcal{T}$. The index ℓ of the sequent is chosen *large enough*, i.e., such that derivability of $C \sqsubseteq_\ell D$ implies derivability of $C \sqsubseteq_\infty D$. In [6] it is shown how an appropriate number ℓ of polynomial size can be computed from the size of the input Γ and $\mathcal{O}$.

Second, to avoid nonterminating runs of the algorithm, a *blocking mechanism* needs to be employed. This mechanism prevents cyclic dependencies between sequents where the derivability of one sequents depends on the derivability of another sequent and vice versa. This problem did not occur in the algorithm for classical unification in [4] due to the fact that, for classical unification, the generation of a cyclic assignment causes the run to fail. For hybrid unification, cyclic assignments may lead to valid hybrid unifiers. In order to realize blocking, we need to keep track of dependencies between sequents. For this reason, we work with *p-sequents* rather than sequents.

We assume without loss of generality that the input unification problem Γ w.r.t. the input ontology $\mathcal{O}$ is flat. Given $\mathcal{O}$ and Γ, the sets At and $\mathrm{At}_{\mathsf{nv}}$ are defined as above.

Definition 12. *A* flat sequent for Γ and $\mathcal{O}$ *is of the form* $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_n D$ *where* $C_1, \ldots, C_m \in \mathrm{At}, D \in \mathrm{At} \cup \{\top\}$*,* $m \geq 0$*, and* $0 \leq n \leq \ell$*. This sequent is called* ground *if no element of* N_{def} *occurs in it. A* p-sequents for Γ and $\mathcal{O}$ *is a pair* $(C \sqsubseteq_n D, P)$ *such that* $\{C \sqsubseteq_n D\} \cup P$ *is a finite set of flat sequents for* Γ *and* $\mathcal{O}$*.*

Intuitively, the p-sequent $(C \sqsubseteq_n D, P)$ says that we need to find a proof tree for $C \sqsubseteq_n D$, and that the proof trees for all the elements of P must contain this proof tree, i.e., the derivations of the elements of P depend on the derivation of $C \sqsubseteq_n D$.

Starting with the initial set of p-sequents

$$\Gamma_p^{(0)} := \{(C \sqsubseteq_\ell D, \emptyset) \mid C \sqsubseteq D \in \Gamma\}$$

the algorithm maintains a current set of p-sequents Γ_p and a current assignment ζ, which initially assigns the empty set to all $X \in N_{def}$. In addition, for each p-sequent in Γ_p it maintains the information on whether it is *solved* or not. Initially, all p-sequents are unsolved, except those with a defined concept on the right-hand side of its first component.[2] Rules are applied only to unsolved p-sequents. A (non-failing) rule application does the following:

- it solves exactly one unsolved p-sequent,
- it may extend the current assignment ζ, and
- it may add new p-sequents to Γ_p, which are marked unsolved unless their first component has a defined concept on the right-hand side.

Adding a new p-sequent is realized through the blocking procedure. This procedures checks whether the new sequent introduces cyclic derivability obligations

[2] Such p-sequents are dealt with by expansion rather than by applying a rule (see below).

Eager Axiom Solving:

Condition: This rule applies to $(\mathfrak{s}, P)$, if $\mathfrak{s}$ is of the form $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_0 D$ or $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_n \top$.
Action: Its application marks $(\mathfrak{s}, P)$ as *solved.*

Eager Ground Solving:

Condition: This rule applies to $(\mathfrak{s}, P)$ with $\mathfrak{s} = C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_n D$, if $\mathfrak{s}$ is ground.
Action: If $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_{\mathcal{T}} D$ does not hold, the rule application fails. Otherwise, $(\mathfrak{s}, P)$ is marked as *solved.*

Eager Solving:

Condition: This rule applies to $(\mathfrak{s}, P)$ with $\mathfrak{s} = C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_n D$, if there is an index $i \in \{1, \ldots, m\}$ such that $C_i = D$ or $C_i = X \in N_{def}$ and $D \in \zeta_X$.
Action: The application marks $(\mathfrak{s}, P)$ as *solved.*

Fig. 2. The eager rules of hybrid unification

(in which case it fails) and whether the sequent to be added already exists (in which case it re-uses the existing sequent, but updates the dependency information). Only if these two cases do not apply does it add the new sequent. To be more precise, given a set of p-sequents Γ_p and a p-sequents $(C \sqsubseteq_n D, P)$, the procedure *blocking* applied to this input does the following:

B1: If the sequent $C \sqsubseteq_n D$ belongs to P, then blocking *fails.*
B2: Otherwise, if there is a p-sequent of the form $(C \sqsubseteq_n D, P')$ in Γ_p, then do the following:
  - Extend the second component of this sequent to $P' \cup P$.
  - For each p-sequent $(\_, P'')$ in Γ_p such that $C \sqsubseteq_n D$ is in P'', extend the second component to $P'' \cup P$,

B3: Otherwise, add $(C \sqsubseteq_n D, P)$ to Γ_p.

Each rule application that extends ζ_X additionally *expands Γ_p w.r.t. X* as follows: every p-sequent of the form $(C_1 \sqcap \cdots \sqcap C_n \sqsubseteq_n X, P)$ is *expanded* by applying blocking to $(C_1 \sqcap \cdots \sqcap C_n \sqsubseteq_{n-1} D, \emptyset)$ and Γ_p for every $D \in \zeta_X$. Since the second components of the p-sequents provided as inputs for blocking are empty, blocking cannot fail during expansion. Note that expansion basically corresponds to an application of the rule (DefR) of HC together with an appropriate number of applications of (AndR).

If a p-sequent $\mathfrak{p}$ is marked as solved, this does not mean that a proof tree for its first component $\mathfrak{s}$ has already been constructed (w.r.t. $\mathcal{O}$ and the TBox induced by the current assignment). It may be the case that the task of constructing the proof tree for $\mathfrak{s}$ was deferred to constructing a proof tree for the first component $\mathfrak{s}'$ of a "smaller" p-sequent. The proof tree for $\mathfrak{s}'$ is then part of the proof tree for $\mathfrak{s}$, and thus $\mathfrak{s}$ needs to be added to the second component of $\mathfrak{p}'$.

The rules of the algorithm consist of three *eager* rules, which are deterministic (see Figure 2), and three *nondeterministic* rules (see Figure 3). Eager rules are applied with higher priority than nondeterministic rules. Among the eager rules,

Decomposition:

Condition: This rule applies to $(\mathfrak{s}, P)$ with $\mathfrak{s} = C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_n \exists s.D'$, if there is a $C_i = \exists s.C'$ such that *blocking* does not fail if applied to $(C' \sqsubseteq_n D', P \cup \{\mathfrak{s}\})$ and Γ_p.
Action: Its application chooses such an index i and applies *blocking* to $(C' \sqsubseteq_n D', P \cup \{\mathfrak{s}\})$ and Γ_p. Once *blocking* was applied, it expands Γ_p w.r.t. D' if $D' \in N_{def}$, and marks $(\mathfrak{s}, P)$ as *solved.*

Extension:

Condition: This rule applies to $(\mathfrak{s}, P)$ with $\mathfrak{s} = C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_n D$ if there is at least one $i \in \{1, \ldots, m\}$ with $C_i \in N_{def}$.
Action: Its application chooses such an index i and adds D to ζ_{C_i}. Γ_p is expanded w.r.t. C_i and $(\mathfrak{s}, P)$ is marked as *solved.*

Mutation:

Condition: This rule applies to $(\mathfrak{s}, P)$ with $\mathfrak{s} = C_1 \sqcap \ldots \sqcap C_m \sqsubseteq_n D$, if there is a GCI $E_1 \sqcap \ldots \sqcap E_k \sqsubseteq F$ in $\mathcal{O}$ and a set $S \subseteq \{1, \ldots, m\}$ such that *blocking* does not fail if applied to Γ_p and each of the p-sequents $(\bigsqcap_{j \in S} C_j \sqsubseteq_n E_1, P \cup \{\mathfrak{s}\}), \ldots, (\bigsqcap_{j \in S} C_j \sqsubseteq_n E_k, P \cup \{\mathfrak{s}\})$, and $(F \sqsubseteq_n D, P \cup \{\mathfrak{s}\})$.
Action: Its application chooses such a GCI $E_1 \sqcap \ldots \sqcap E_k \sqsubseteq F$ and a set $S \subseteq \{1, \ldots, m\}$. It applies *blocking* to Γ_p and each of the p-sequents $(\bigsqcap_{j \in S} C_j \sqsubseteq_n E_1, P \cup \{\mathfrak{s}\}), \ldots, (\bigsqcap_{j \in S} C_j \sqsubseteq_n E_k, P \cup \{\mathfrak{s}\})$, and $(F \sqsubseteq_n D, P \cup \{\mathfrak{s}\})$. Once *blocking* was applied, $(\mathfrak{s}, P)$ is marked as *solved.*

Fig. 3. The nondeterministic rules of hybrid unification

Eager Axiom Solving has the highest priority, then comes Eager Ground Solving, and then Eager Solving.

Algorithm 13. Let Γ w.r.t. $\mathcal{O}$ be a flat $\mathcal{EL}$-unification problem. We set $\Gamma_p := \Gamma_p^{(0)}$ and $\zeta_X := \emptyset$ for all $X \in N_{def}$. While Γ_p contains an unsolved p-sequent, apply the steps (1) and (2).

(1) **Eager rule application:** If some eager rules apply to an unsolved p-sequent $\mathfrak{p}$ in Γ_p, apply one of highest priority. If the rule application fails, then return "no hybrid unifier".
(2) **Nondeterministic rule application:** If no eager rule is applicable, let $\mathfrak{p}$ be an unsolved p-sequent in Γ_p. If one of the nondeterministic rules applies to $\mathfrak{p}$, nondeterministically choose one of these rules and apply it. If none of these rules apply to $\mathfrak{p}$, then return "no hybrid unifier".

Once all p-sequents are solved, return the TBox $\mathcal{T}$ induced by the current assignment.

In step (2), the choice which unsolved p-sequent to consider next is don't care nondeterministic. However, choosing which rule to apply to the chosen p-sequent is don't know nondeterministic. Additionally, the application of nondeterministic rules requires don't know nondeterministic guessing.

The *eager rules* are mainly there for optimization purposes, i.e., to avoid nondeterministic choices if a deterministic decision can easily be made. For example, given a ground sequent $C \sqsubseteq_n D$, as considered in the *Eager Ground Solving* rule, the GCI $C \sqsubseteq D$ either follows from the ontology $\mathcal{O}$, in which case any TBox is a hybrid unifier of it, or it does not, in which case there is no hybrid unifier. This condition can be checked in polynomial time since subsumption w.r.t. hybrid $\mathcal{EL}$-ontologies is polynomial [11,14,13]. In the case considered in the *Eager Solving* rule, the TBox induced by the current assignment obviously already implies the GCI $C_1 \sqcap \ldots \sqcap C_m \sqsubseteq D$. The *Eager Axiom Solving* rule corresponds to the rules (Top) and (Start) of HC. Note that the rule (Refl) of HC is covered by *Eager Solving.*

The *nondeterministic rules* only come into play if no eager rules can be applied. In order to solve an unsolved p-sequent $(\mathfrak{s}, P)$, one considers which rule of HC could have been applied to obtain $\mathfrak{s}$. The rules *Extension* and *Decomposition* respectively correspond to applications of rules (DefL) and (Ex) of HC, together with an appropriate number of applications of the rules (AndLi). The *Mutation* rule corresponds to an application of the (GCI) rule from HC, again together with an appropriate number of applications of the rules (AndLi).

Due to the space restrictions, we cannot give details on how to prove that the algorithm is correct. Complete proofs of soundness, completeness and termination can be found in [6].

Theorem 14. *Algorithm 13 is an* NP*-decision procedure for hybrid $\mathcal{EL}$-unifiability w.r.t. arbitrary $\mathcal{EL}$-ontologies.*

6 Conclusions

In this paper, we have first proved that hybrid $\mathcal{EL}$-unification w.r.t. arbitrary $\mathcal{EL}$-ontologies is NP-complete, and then developed a goal-oriented NP-algorithm for hybrid $\mathcal{EL}$-unification that is better than the brute-force "guess and then test" algorithm used to show the "in NP" result. As illustrated by Example 6, computing hybrid unifiers rather than classical ones may be appropriate in some situations. Nevertheless, the decidability and complexity of classical $\mathcal{EL}$-unification w.r.t. arbitrary $\mathcal{EL}$-ontologies is an important topic for future research. We hope that hybrid unification may also be helpful in this context. Basically, given a hybrid unifier $\mathcal{T}$ of Γ w.r.t. $\mathcal{O}$, we can obtain a classical unifier of Γ w.r.t. $\mathcal{O}$ by finding an acyclic TBox $\mathcal{S}$ such that $\mathcal{O} \cup \mathcal{S}$ entails all the GCIs that $(\mathcal{O}, \mathcal{T})$ entails w.r.t. hybrid semantics, i.e. $C \sqsubseteq_{gfp,\mathcal{O},\mathcal{T}} D$ implies $C \sqsubseteq_{\mathcal{O} \cup \mathcal{S}} D$ for all (relevant) concept descriptions C, D.

References

1. Baader, F.: Terminological cycles in a description logic with existential restrictions. In: Gottlob, G., Walsh, T. (eds.) Proc. of the 18th Int. Joint Conf. on Artificial Intelligence (IJCAI 2003), pp. 325–330. Morgan Kaufmann, Los Altos (2003)
2. Baader, F., Borgwardt, S., Morawska, B.: Unification in the description logic $\mathcal{EL}$ w.r.t. cycle-restricted TBoxes. LTCS-Report 11-05, Chair for Automata Theory, Institute for Theoretical Computer Science, Technische Universität Dresden, Dresden, Germany (2011), `http://lat.inf.tu-dresden.de/research/reports.html`

3. Baader, F., Borgwardt, S., Morawska, B.: Extending unification in $\mathcal{EL}$ towards general TBoxes. In: Proc. of the 13th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2012), pp. 568–572. AAAI/MIT Press (2012)
4. Baader, F., Borgwardt, S., Morawska, B.: A goal-oriented algorithm for unification in $\mathcal{ELH}_{R+}$ w.r.t. Cycle-restricted ontologies. In: Thielscher, M., Zhang, D. (eds.) AI 2012. LNCS, vol. 7691, pp. 493–504. Springer, Heidelberg (2012)
5. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
6. Baader, F., Fernández Gil, O., Morawska, B.: Hybrid unification in the description logic $\mathcal{EL}$. LTCS-Report 13-07, Chair for Automata Theory, Institute for Theoretical Computer Science, Technische Universität Dresden, Dresden, Germany (2013), `http://lat.inf.tu-dresden.de/research/reports.html`
7. Baader, F., Morawska, B.: Unification in the description logic $\mathcal{EL}$. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 350–364. Springer, Heidelberg (2009)
8. Baader, F., Morawska, B.: Unification in the description logic $\mathcal{EL}$. Logical Methods in Computer Science 6(3) (2010)
9. Baader, F., Narendran, P.: Unification of concept terms in description logics. J. of Symbolic Computation 31(3), 277–305 (2001)
10. Brandt, S.: Polynomial time reasoning in a description logic with existential restrictions, GCI axioms, and—what else? In: de Mántaras, R.L., Saitta, L. (eds.) Proc. of the 16th Eur. Conf. on Artificial Intelligence (ECAI 2004), pp. 298–302 (2004)
11. Brandt, S., Model, J.: Subsumption in $\mathcal{EL}$ w.r.t. hybrid tboxes. In: Furbach, U. (ed.) KI 2005. LNCS (LNAI), vol. 3698, pp. 34–48. Springer, Heidelberg (2005)
12. Küsters, R.: Non-Standard Inferences in Description Logics. LNCS (LNAI), vol. 2100. Springer, Heidelberg (2001)
13. Novaković, N.: Proof-theoretic Approach to Deciding Subsumption and Computing Least Common Subsumer in EL w.r.t. Hybrid TBoxes. Master's thesis, Chair for Automata Theory, Institute for Theoretical Computer Science, Technische Universität Dresden, Germany (2007), `http://lat.inf.tu-dresden.de/research/mas/#Nov-Mas-07`
14. Novaković, N.: A proof-theoretic approach to deciding subsumption and computing least common subsumer in $\mathcal{EL}$ w.r.t. hybrid TBoxes. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 311–323. Springer, Heidelberg (2008)

Disproving Confluence of Term Rewriting Systems by Interpretation and Ordering

Takahito Aoto

RIEC, Tohoku University,
2-1-1 Katahira, Aoba-ku, Sendai 980-8577, Japan
aoto@nue.riec.tohoku.ac.jp

Abstract. In order to disprove confluence of term rewriting systems, we develop new criteria for ensuring non-joinability of terms based on interpretation and ordering. We present some instances of the criteria which are amenable for automation, and report on an implementation of a confluence disproving procedure based on these instances. The experiments reveal that our method is successfully applied to automatically disprove confluence of some term rewriting systems, on which state-of-the-art automated confluence provers fail. A key idea to make our method effective is the introduction of usable rules—this allows one to decompose the constraint on rewrite rules into smaller components that depend on starting terms.

Keywords: Confluence, Non-Joinability, Interpretation, Ordering, Term Rewriting Systems.

1 Introduction

Confluence is a property that often turns out to be useful in vast topics of term rewriting; hence, it is conceived as one of the central properties of term rewriting (see e.g. [5,30]). There is conceivably a long history for the development of techniques for proving confluence of term rewriting systems (TRSs, for short); see e.g. [5,30,31]. Recently, the area of proving confluence of TRSs *automatically* also caught an increasing attention. Indeed, recent works on confluence proving often address also automation of the methods [1,2,19,21,34]. Furthermore, several implementations of confluence provers are emerging [3,17,33], and the first competition on confluence provers (CoCo 2012) has been held last year.

For automated confluence provers, it is also important to *disprove* confluence so that they can give up unsuccessful attempts for confluence proving. In contrast to many dedicated techniques for proving confluence, however, not many techniques for disproving confluence are known. A typical approach to disprove confluence of (non-terminating) TRSs is first to construct a candidate of two terms that can be reduced from a common term, and then to show that they are not joinable, i.e. two terms do not have a common reduct. In this scenario, as well as the selection of the candidates, proving *non-joinability* of terms is

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 311–326, 2013.

essential. So far, the only serious approach to prove the non-joinability of terms is to use approximation by tree automata [9,12][1], implemented in CSI [33].

In this paper, we give new methods for proving that given two terms s, t are not joinable. The first method consists in giving an *interpretation*, e.g. a mapping from terms to natural numbers, that is preserved by the application of usable rules and such that the interpretation of s is different from that of t. The second method consists in giving an *ordering* $>$ such that $s > t$, and usable rules from s are non-decreasing and the usable rules from t are non-increasing. These methods are implemented using polynomial interpretations and recursive path orderings—interpretations and orderings that are widely used in the literature for termination proving. The experiments reveal that our methods can be applied to automatically disprove confluence of some term rewriting systems, on which state-of-the-art automated confluence provers fail.

The rest of the paper is organized as follows. In Section 2, we give some basic definitions and fix some notations to be used in the paper. In Section 3, we give an abstract non-joinability criterion using interpretation; the criterion is extended in Section 4 by a notion of usable rules for reachability. In Section 5, we give non-joinability criteria using ordering in terms of interpretation and rewrite relation; the latter is extended in Section 6 incorporating the notion of argument filtering from the area of termination proving. Related works are discussed in Section 7. In Section 8, we report on our implementation and experiments. Section 9 concludes.

2 Preliminaries

The *product* of two sets A and B is denoted by $A \times B$. We write A^2 for $A \times A$, and more generally, A^n (for the n-fold product of A) or $\Pi_{i \in I} A_i$, where I is an arbitrary index set. A *tuple* of elements is denoted by $\langle a_1, \ldots, a_n \rangle$ or $\langle a_i \rangle_{i \in I}$. The *disjoint union* of two sets A and B is denoted by $A \uplus B$, and that of all A_i $(i \in I)$ by $\biguplus_{i \in I} A_i$. The *composition* of relations R and S is denoted by $R \circ S$. The *reflexive transitive closure* of R is denoted by R^*; for a relation $\rightarrow$, its *reverse* is denoted by $\leftarrow$, the *reflexive closure* by $\xrightarrow{=}$ and the *reflexive transitive closure* by $\xrightarrow{*}$. We write $a_0 \rightarrow^n a_n$ to denote $a_0 \rightarrow a_1 \rightarrow \cdots \rightarrow a_n$ and $a \rightarrow^{\leq n} b$ if $a \rightarrow^m b$ for some $m \leq n$. We say an element a is a *normal form* (w.r.t. $\rightarrow$) if there exists no b such that $a \rightarrow b$. The relation $\rightarrow$ is *well-founded* if there exists no infinite chain $a_0 \rightarrow a_1 \rightarrow \cdots$. A relation is a *partial order* if it is reflexive, transitive and antisymmetric, and a *quasi-order* if it is reflexive and transitive.

We consider arity-fixed function symbols. The *arity* of function symbol f is denoted by $\mathrm{arity}(f)$. Function symbol f is a *constant* if $\mathrm{arity}(f) = 0$. The set of terms over a set $\mathcal{F}$ of function symbols and the set $\mathcal{V}$ of variables is denoted by $\mathrm{T}(\mathcal{F}, \mathcal{V})$. Terms which are not variables are referred to as *non-variable* terms and terms containing no variables are called *ground* terms. The set of variables in a term t is denoted by $\mathcal{V}(t)$. Let $\Box$ (called a *hole*) be a special constant that

[1] The technique is investigated in different literature. Applications of the technique are found also in the literature for termination proving [23,25].

is not involved in $\mathcal{F}$. A *context* is a term in $\mathrm{T}(\mathcal{F} \cup \{\Box\}, \mathcal{V})$ that contains exactly one hole in the term. The term in $\mathrm{T}(\mathcal{F}, \mathcal{V})$ obtained by replacing the hole in a context C with a term t is denoted by $C[t]$. A context C is *empty* if $C = \Box$ and *non-empty* otherwise. A term s is said to be a *subterm* of t if $t = C[s]$ for some context C. We write $s \trianglelefteq t$ to denote that s is a subterm of t. The set of *positions* in a term t is denoted by $\mathrm{Pos}(t)$. A term t can be identified with a mapping $\mathrm{Pos}(t) \to \mathcal{F}$. The *root* position is denoted by ϵ; thus the root symbol of a term t is denoted by $t(\epsilon)$. The subterm at the position $p \in \mathrm{Pos}(t)$ is denoted by $t|_p$. We write $C[s]_p$ if $C(p) = \Box$.

A *substitution* is a mapping $\mathcal{V} \to \mathrm{T}(\mathcal{F}, \mathcal{V})$, which is homomorphically extended to the mapping $\mathrm{T}(\mathcal{F}, \mathcal{V}) \to \mathrm{T}(\mathcal{F}, \mathcal{V})$. For any substitution θ and term t, $\theta(t)$ is written as $t\theta$ if no confusion arises. Terms s and t are said to be *unifiable* if $s\theta = t\theta$ for some substitution θ. We write $\mathrm{Unif}(s, t)$ to denote that the terms s and t are unifiable.

A *rewrite rule* $l \to r$ is a pair of terms[2]. A *term rewriting system* (*TRS*, for short) is a set of rewrite rules. For a TRS $\mathcal{R}$, a *rewrite step* $s \to_{\mathcal{R}} t$ is given if $s = C[l\theta]$ and $t = C[r\theta]$ for some rewrite rule $l \to r \in \mathcal{R}$, context C and substitution θ. A relation R on terms is said to be *closed under contexts* if $s\ R\ t$ implies $C[s]\ R\ C[t]$ for any context C, is *closed under substitutions* if $s\ R\ t$ implies $s\theta\ R\ t\theta$ for any substitution θ. A *rewrite relation* is a relation on terms that is closed under contexts and substitutions. Given a TRS $\mathcal{R}$, it is readily checked that the relation $\to_{\mathcal{R}} = \{\langle s, t\rangle \in \mathrm{T}(\mathcal{F}, \mathcal{V})^2 \mid s \to_{\mathcal{R}} t\}$ is a rewrite relation, and is said to be the rewrite relation of $\mathcal{R}$. If no confusion arises, the subscript of $\to_{\mathcal{R}}$ will be omitted. A partial order (quasi-order) is a *rewrite* partial order (*rewrite* quasi-order, respectively) if it is a rewrite relation.

Given a term s, the sets of terms $\{t \in \mathrm{T}(\mathcal{F}, \mathcal{V}) \mid s \xrightarrow{*} t\}$ and $\{t \in \mathrm{T}(\mathcal{F}, \mathcal{V}) \mid t \xrightarrow{*} s\}$ are denoted by $[s](\xrightarrow{*})$ and $(\xrightarrow{*})[s]$, respectively. Terms s and t are said to be *joinable* if $[s](\xrightarrow{*}) \cap [t](\xrightarrow{*}) \neq \emptyset$, and *non-joinable* otherwise. We write $\mathrm{NJ}(s, t)$ to denote that the terms s and t are non-joinable. A TRS $\mathcal{R}$ is *confluent* if for any terms s, t, $(\xrightarrow{*})[s] \cap (\xrightarrow{*})[t] \neq \emptyset$ implies $[s](\xrightarrow{*}) \cap [t](\xrightarrow{*}) \neq \emptyset$. A TRS $\mathcal{R}$ is *terminating* if $\to_{\mathcal{R}}$ is well-founded. It is known that confluence of a TRS $\mathcal{R}$ is decidable if $\mathcal{R}$ is terminating.

In order to disprove that a (non-terminating) TRS $\mathcal{R}$ is confluent, we construct two terms s and t such that $(\xrightarrow{*})[s] \cap (\xrightarrow{*})[t] \neq \emptyset$ in some way, and then prove $\mathrm{NJ}(s, t)$. In order to check non-joinability of terms, it suffices to check ground instances of them using fresh constants [33]. From here on, we concentrate on the problem of proving non-joinability of ground terms.

3 Proving Non-joinability by Interpretation

In this section, we give an abstract criterion to prove non-joinability of terms based on their interpretations in $\mathcal{F}$-algebras. Then we point out why it is not useful for our purpose—we will fix the problem in the next section.

[2] Here, we drop the usual restriction that $l \notin \mathcal{V}$ and $\mathcal{V}(r) \subseteq \mathcal{V}(l)$; we will deal with rewrite rules that do not satisfy these restrictions in Section 6.

We first recall some basic terminology on semantics of the equational logic and fix our notations (see e.g. [5]). An *$\mathcal{F}$-algebra* $\mathcal{A} = \langle A, \langle f^{\mathcal{A}} \rangle_{f \in \mathcal{F}} \rangle$ is a pair of a set A and a tuple of functions $f^{\mathcal{A}} : A^n \to A$ for each n-ary function symbol $f \in \mathcal{F}$; the set A is called the *carrier set* of the $\mathcal{F}$-algebra $\mathcal{A}$. A *valuation* on $\mathcal{A}$ is a mapping $\mathcal{V} \to A$. The *interpretation* $[\![t]\!]_{\mathcal{A},\sigma}$ (which is abbreviated as $[\![t]\!]_\sigma$ for brevity) of a term $t \in \mathrm{T}(\mathcal{F}, \mathcal{V})$ w.r.t. a valuation σ on $\mathcal{A}$ is recursively defined by $[\![x]\!]_\sigma = \sigma(x)$ and $[\![f(t_1, \ldots, t_n)]\!]_\sigma = f^{\mathcal{A}}([\![t_1]\!]_\sigma, \ldots, [\![t_n]\!]_\sigma)$. For any substitution θ and valuation σ, a valuation $[\![\theta]\!]_\sigma$ is given by $[\![\theta]\!]_\sigma(x) = [\![\theta(x)]\!]_\sigma$. We note that the interpretation of ground terms is independent of valuation, i.e., for any ground term t, $[\![t]\!]_\sigma = [\![t]\!]_\rho$ holds for any valuations σ, ρ. Hence, w.l.o.g. we drop valuations to denote the interpretation of ground terms.

The next property is well-known (e.g. [5]).

Lemma 1. *Fix an $\mathcal{F}$-algebra $\mathcal{A}$. For any term t, substitution θ and valuation σ, $[\![t\theta]\!]_\sigma = [\![t]\!]_{(\sigma \circ [\![\theta]\!]_\sigma)}$.* □

Using the notion of interpretation of terms in $\mathcal{F}$-algebras, the following criterion for non-joinability of (ground) terms naturally arises.

Theorem 2. *Let s, t be ground terms and $\mathcal{A} = \langle A, \langle f^{\mathcal{A}} \rangle_{f \in \mathcal{F}} \rangle$ an $\mathcal{F}$-algebra such that $A = \biguplus_{i \in I} A_i$. Suppose (i) for any valuation σ and $l \to r \in \mathcal{R}$, if $[\![l]\!]_\sigma \in A_i$ then $[\![r]\!]_\sigma \in A_i$, (ii) for any $f \in \mathcal{F}$, $a \in A$ and $i, j \in I$, if $a \in A_i$ implies $f^{\mathcal{A}}(\ldots, a, \ldots) \in A_j$, then $f^{\mathcal{A}}(\ldots, b, \ldots) \in A_j$ for any $b \in A_i$ and (iii) $[\![s]\!] \in A_i$ and $[\![t]\!] \in A_j$ with $i \neq j$. Then* $\mathrm{NJ}(s, t)$.

Proof. It is straightforward to show by induction on C that for any valuation σ and $l \to r \in \mathcal{R}$, $[\![C[l\theta]]\!]_\sigma \in A_i$ implies $[\![C[r\theta]]\!]_\sigma \in A_i$. Thus for any valuation σ, $u \to_{\mathcal{R}} v$ and $[\![u]\!]_\sigma \in A_i$ imply $[\![v]\!]_\sigma \in A_i$. Suppose that $\mathrm{NJ}(s, t)$ does not hold, i.e. $u \in [s](\xrightarrow{*}) \cap [t](\xrightarrow{*})$ for some u. Then we obtain $[\![u]\!]_\sigma \in A_i$ from $[\![s]\!]_\sigma \in A_i$ and $[\![u]\!]_\sigma \in A_j$ from $[\![t]\!]_\sigma \in A_j$. This contradicts our assumption that $A_i \cap A_j = \emptyset$. □

Although Theorem 2 may be applied to prove non-joinability of two terms in general, it is not effective in our setting—in the context of disproving confluence, one wants to show $\mathrm{NJ}(s, t)$ for s, t satisfying $(\xrightarrow{*})[s] \cap (\xrightarrow{*}])[t] \neq \emptyset$. For such s, t, if the conditions (i), (ii) of the theorem are satisfied, then $s \in A_i \Leftrightarrow t \in A_i$ holds, and hence the condition (iii) never holds.

The trick to apply the idea in our setting is to relax the condition (i) so that the constraint is applied not to all rules but only to rules usable in the reductions starting from s or t, which will be explored in the next section.

4 Usable Rules for Reachability

In the literature for proving termination of TRSs, a notion of usable rules is known to be very useful in the dependency pairs technique [4,14,18,32]. There, the name 'usable' originally comes from the fact that usable rules are rules possibly used to connect dependency pairs. Naturally, it is not very suitable for our purpose, because, in our setting, usable rules are to be the collection of rules that are possibly used in reductions from an initial term.

To suit our setting, we introduce a notion of *usable rules for reachability.* For this, the notion of TCAP [13] (introduced for defining usable rules for dependency pairs) is helpful. For terms t, $\mathrm{TCAP}(t)$ is defined recursively by: $\mathrm{TCAP}(x) = x'$, $\mathrm{TCAP}(f(t_1,\ldots,t_n)) = x'$ if $\mathrm{Unif}(f(u_1,\ldots,u_n),l)$ for some $l \to r \in \mathcal{R}$, and $\mathrm{TCAP}(f(t_1,\ldots,t_n)) = f(u_1,\ldots,u_n)$ otherwise, where $u_i = \mathrm{TCAP}(t_i)$ $(1 \leq i \leq \mathrm{arity}(f))$. Here, a new fresh variable is taken for x' every time it is used.

Definition 3 (usable rules for reachability). *For any TRS $\mathcal{R}$ and term s, let $\mathcal{U}_0(\mathcal{R},s)$ be the smallest set $\mathcal{U}_0(\mathcal{R},s) \subseteq \mathcal{R}$ satisfying the following conditions: (i) if $l \to r \in \mathcal{R}$ with $l \in \mathcal{V}$, then $l \to r \in \mathcal{U}_0(\mathcal{R},s)$; (ii) for any $l \to r \in \mathcal{R}$ and non-variable subterm $f(u_1,\ldots,u_n) \trianglelefteq s$, if* $\mathrm{Unif}(f(\mathrm{TCAP}(u_1),\ldots,\mathrm{TCAP}(u_n)),l)$ *then $l \to r \in \mathcal{U}_0(\mathcal{R},s)$; (iii) if $l' \to r' \in \mathcal{U}_0(\mathcal{R},s)$ and $l \to r \in \mathcal{U}_0(\mathcal{R},r')$, then $l \to r \in \mathcal{U}_0(\mathcal{R},s)$. The set $\mathcal{U}_{\mathsf{r}}(\mathcal{R},s)$ of* usable rules for reachability *w.r.t. a TRS $\mathcal{R}$ and a term s is defined by $\mathcal{U}_{\mathsf{r}}(\mathcal{R},s) = \mathcal{R}$ if there exists $l \to r \in \mathcal{U}_0(\mathcal{R},s)$ such that $\mathcal{V}(r) \not\subseteq \mathcal{V}(l)$ and $\mathcal{U}_{\mathsf{r}}(\mathcal{R},s) = \mathcal{U}_0(\mathcal{R},s)$ otherwise.*

Under the usual variable restrictions on rewrite rules, our notion of usable rules corresponds to the one obtained from the usable rules for innermost termination [13] by replacing ICAP with TCAP. (Standard) usable rules $\mathcal{U}(\mathcal{R},s)$ in dependency pairs [4] is different from $\mathcal{U}_{\mathsf{r}}(\mathcal{R},s)$—for example, for $\mathcal{R} = \{\mathsf{f}(\mathsf{a}) \to \mathsf{b}\}$, we have $\mathcal{U}_{\mathsf{r}}(\mathcal{R},\mathsf{f}(\mathsf{b})) = \emptyset \neq \mathcal{R} = \mathcal{U}(\mathcal{R},\mathsf{f}(\mathsf{b}))$. Under the usual restriction of rewrite rules on variables, it is easy to check $\mathcal{U}_{\mathsf{r}}(\mathcal{R},s) \subseteq \mathcal{U}(\mathcal{R},s)$.

The following is a key lemma to prove our theorem below.

Lemma 4. *Let $\mathcal{R}$ be a TRS, $l \to r \in \mathcal{R}$ and s,t terms. If $s \xrightarrow{*}_{\mathcal{R}} \circ \to_{\{l \to r\}} t$ then $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R},s)$.*

Proof. The proof consists of proving the following three claims step by step.

1. If $s \to_{\{l \to r\}} t$ then $l \to r \in \mathcal{U}_0(\mathcal{R},s)$ (and hence, $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R},s)$).
2. If $s \to_{\{l \to r\}} t$ then $\mathcal{U}_{\mathsf{r}}(\mathcal{R},t) \subseteq \mathcal{U}_{\mathsf{r}}(\mathcal{R},s)$.
3. If $s \xrightarrow{*}_{\mathcal{R}} \circ \to_{\{l \to r\}} t$ then $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R},s)$.

1. Suppose $s \to_{\{l \to r\}} t$. If $l \in \mathcal{V}$ then $l \to r \in \mathcal{U}_0(\mathcal{R},s)$ by definition. Otherwise, there exists non-variable subterm $u \trianglelefteq s$ such that $u = f(u_1,\ldots,u_n) = l\theta$ for some substitution θ. Then, as $u_i = \mathrm{TCAP}(u_i)\theta_i$ for some θ_i, we have $f(\mathrm{TCAP}(u_1)\theta_1,\ldots,\mathrm{TCAP}(u_n)\theta_n) = l\theta$. Because one can assume w.l.o.g. that $\mathcal{V}(\mathrm{TCAP}(u_i)) \cap \mathcal{V}(\mathrm{TCAP}(u_j)) = \emptyset$ for $i \neq j$ and $\mathcal{V}(\mathrm{TCAP}(u_i)) \cap \mathcal{V}(l) = \emptyset$, it follows that $f(\mathrm{TCAP}(u_1),\ldots,\mathrm{TCAP}(u_n))$ and l are unifiable, and hence $l \to r \in \mathcal{U}_0(\mathcal{R},s)$.
2. Since $l \to r \in \mathcal{U}_0(\mathcal{R},s)$ by the claim 1, if $\mathcal{V}(r) \not\subseteq \mathcal{V}(l)$ then $\mathcal{U}_{\mathsf{r}}(\mathcal{R},s) = \mathcal{R} \supseteq \mathcal{U}_{\mathsf{r}}(\mathcal{R},t)$. So, suppose $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. It suffices to show $\mathcal{U}_0(\mathcal{R},t) \subseteq \mathcal{U}_0(\mathcal{R},s)$. We show the claim by induction on the definition of $\mathcal{U}_0(\mathcal{R},t)$. Suppose $l' \to r' \in \mathcal{U}_0(\mathcal{R},t)$. (i) Suppose $l' \in V$. Then by definition $l' \to r' \in \mathcal{U}_0(\mathcal{R},s)$.

(ii) Suppose there exists a non-variable subterm $u = f(u_1, \ldots, u_n) \trianglelefteq t$ such that $\mathrm{Unif}(f(\mathrm{TCAP}(u_1), \ldots, \mathrm{TCAP}(u_n)), l')$. Let $t = C[r\theta]$. Then either (a) $u \trianglelefteq \theta(x)$ for some $x \in \mathcal{V}(r)$, (b) $u \trianglelefteq C$, (c) $u = v\theta$ for some non-variable subterm $v = f(v_1, \ldots, v_n) \trianglelefteq r$ or (d) $u = C'[r\theta]$ for some non-empty context $C' \trianglelefteq C$. In the cases of (a) and (b), because $\mathcal{V}(r) \subseteq \mathcal{V}(l)$, we have $u \trianglelefteq s = C[l\theta]$ and hence $l' \to r' \in \mathcal{U}_0(\mathcal{R}, s)$ by definition. In the case of (c), by $\mathrm{Unif}(f(\mathrm{TCAP}(u_1), \ldots, \mathrm{TCAP}(u_n)), l')$ and $f(u_1, \ldots, u_n) = f(v_1, \ldots, v_n)\theta$, we have $\mathrm{Unif}(f(\mathrm{TCAP}(v_1), \ldots, \mathrm{TCAP}(v_n)), l')$. Hence because $v = f(v_1, \ldots, v_n) \trianglelefteq r$, it follows $l' \to r' \in \mathcal{U}_0(\mathcal{R}, r)$. Thus, since $l \to r \in \mathcal{U}_0(\mathcal{R}, s)$ by our claim 1, we have $l' \to r' \in \mathcal{U}_0(\mathcal{R}, s)$ by definition. In the case of (d), let $C' = f(u_1, \ldots, \tilde{C}, \ldots, u_n)$. Then because of $\mathrm{Unif}(f(\mathrm{TCAP}(u_1), \ldots, \mathrm{TCAP}(\tilde{C}[r\theta]), \ldots, \mathrm{TCAP}(u_n)), l')$, it follows that $\mathrm{Unif}(f(\mathrm{TCAP}(u_1), \ldots, \mathrm{TCAP}(\tilde{C}[x']), \ldots, \mathrm{TCAP}(u_n)), l')$, and thus we have $\mathrm{Unif}(f(\mathrm{TCAP}(u_1), \ldots, \mathrm{TCAP}(\tilde{C}[l\theta]), \ldots, \mathrm{TCAP}(u_n)), l')$. Thus $l' \to r' \in \mathcal{U}_0(\mathcal{R}, s)$ by definition. (iii) Suppose there exists $l'' \to r'' \in \mathcal{U}_0(\mathcal{R}, t)$ and $l' \to r' \in \mathcal{U}_0(\mathcal{R}, r'')$. Then, by induction hypothesis, $l'' \to r'' \in \mathcal{U}_0(\mathcal{R}, s)$ and hence $l' \to r' \in \mathcal{U}_0(\mathcal{R}, s)$ by definition.

3. We show the claim by induction on the length k of $s \xrightarrow{*}_{\mathcal{R}} t$. (B.S.) $k = 1$. Then $s \to_{\{l \to r\}} t$ and thus $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)$ by the claim 1. (I.S.) $k > 1$. Suppose $s \to_{\mathcal{R}} s' \xrightarrow{*}_{\mathcal{R}} \circ \to_{\{l \to r\}} t$. Then by induction hypothesis $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s')$. Since $\mathcal{U}_{\mathsf{r}}(\mathcal{R}, s') \subseteq \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)$ by the claim 2, we obtain $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)$. □

Now, Theorem 2 in the previous section is refined by replacing "$l \to r \in \mathcal{R}$" with "$l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s) \cup \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$."

Theorem 5. *Let s, t be ground terms and $\mathcal{A} = \langle A, \langle f^{\mathcal{A}} \rangle_{f \in \mathcal{F}} \rangle$ an $\mathcal{F}$-algebra such that $A = \biguplus_{i \in I} A_i$. Suppose (i) for any valuation σ and $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s) \cup \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$, if $[\![l]\!]_\sigma \in A_i$ then $[\![r]\!]_\sigma \in A_i$, (ii) for any $f \in \mathcal{F}$, $a \in A$ and $i, j \in I$, if $a \in A_i$ implies $f^{\mathcal{A}}(\ldots, a, \ldots) \in A_j$, then $f^{\mathcal{A}}(\ldots, b, \ldots) \in A_j$ for any $b \in A_i$ and (iii) $[\![s]\!] \in A_i$ and $[\![t]\!] \in A_j$ with $i \neq j$. Then* $\mathrm{NJ}(s, t)$.

Proof. Similar to the proof of Theorem 2, using Lemma 4. □

The criterion of Theorem 5, in general, is not amenable for automation, and one has to use more concrete instances of the theorem such as given below.

Corollary 6. *Let $\mathcal{A}$ be an $\mathcal{F}$-algebra and s, t be ground terms. Suppose (i) $[\![l]\!]_\sigma = [\![r]\!]_\sigma$ for any valuation σ and $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s) \cup \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$ and (ii) $[\![s]\!] \neq [\![t]\!]$. Then* $\mathrm{NJ}(s, t)$. □

Proof. Take the carrier set A itself as the index set and the singleton set $\{a\}$ as A_a for each $a \in A$. □

Corollary 7. *Let s, t be ground terms and $\mathcal{A}$ an $\mathcal{F}$-algebra whose carrier set is a set of integers. Suppose there exists an integer $k \geq 2$ such that (i) for any valuation σ and $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s) \cup \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$, $[\![l]\!]_\sigma \equiv [\![r]\!]_\sigma \pmod{k}$ and (ii) $[\![s]\!] \not\equiv [\![t]\!] \pmod{k}$. Then* $\mathrm{NJ}(s, t)$. □

Proof. Take $I = \{0, 1, \ldots, k-1\}$ and $A_i = \{n \in A \mid n \bmod k = i\}$ for each $i \in I$ and use Theorem 2. □

One way to automate non-joinability check using (instances of) Corollaries 6 and 7 is to use linear polynomial interpretations [5]: Take the set of integers as the carrier set, and for each n-ary function symbol $f \in \mathcal{F}$, let $f^{\mathcal{A}}(x_1, \ldots, x_n) = a_{f,0} + a_{f,1}x_1 + \cdots + a_{f,n}x_n$ where $a_{f,0}, \ldots, a_{f,n}$ are selected from a finite range of integers. Then the criteria of Corollaries 6 and 7 can be encoded as constraint solving problems assigning suitable values for each $a_{f,i}$ ($f \in \mathcal{F}, 0 \leq i \leq \text{arity}(f)$). Indeed, this kind of constraint solving for polynomial interpretations is commonly used in termination tools, and its automation techniques are widely known (e.g. [7,14]).

In following examples, non-confluence is shown using these corollaries.

Example 8. Let

$$\mathcal{R} = \left\{ \begin{array}{llll} (1) & \mathsf{a} \to \mathsf{h}(\mathsf{c}), & (2) & \mathsf{a} \to \mathsf{h}(\mathsf{f}(\mathsf{c})) \\ (3) & \mathsf{h}(x) \to \mathsf{h}(\mathsf{h}(x)), & (4) & \mathsf{f}(x) \to \mathsf{f}(\mathsf{g}(x)) \end{array} \right\}.$$

Let $s = \mathsf{h}(\mathsf{c})$ and $t = \mathsf{h}(\mathsf{f}(\mathsf{c}))$. As $\mathsf{a} \in (\xrightarrow{*})[s] \cap (\xrightarrow{*})[t]$, it suffices to show $\mathrm{NJ}(s,t)$ to disprove the confluence of $\mathcal{R}$. We have $\mathcal{U}_{\mathsf{r}}(\mathcal{R}, s) \cup \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t) = \{(3), (4)\}$. Take an $\mathcal{F}$-algebra $\mathcal{A} = \langle \{0,1\}, \langle f^{\mathcal{A}} \rangle_{f \in \mathcal{F}} \rangle$ as $\mathsf{a}^{\mathcal{A}} = \mathsf{c}^{\mathcal{A}} = 0$, $\mathsf{f}^{\mathcal{A}}(n) = 1 - n$, $\mathsf{h}^{\mathcal{A}}(n) = \mathsf{g}^{\mathcal{A}}(n) = n$. Then for any valuation σ, we have $[\![\mathsf{h}(x)]\!]_\sigma = \sigma(x) = [\![\mathsf{h}(\mathsf{h}(x))]\!]_\sigma$ and $[\![\mathsf{f}(x)]\!]_\sigma = 1 - \sigma(x) = [\![\mathsf{f}(\mathsf{g}(x))]\!]_\sigma$; thus, $[\![l]\!]_\sigma = [\![r]\!]_\sigma$ for each $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s) \cup \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$. Thus $[\![s]\!] = [\![\mathsf{h}(\mathsf{c})]\!] = 0 \neq 1 = [\![t]\!] = [\![\mathsf{h}(\mathsf{f}(\mathsf{c}))]\!]$. Therefore, $\mathrm{NJ}(s,t)$ by Corollary 6.

Example 9. Let

$$\mathcal{R} = \left\{ \begin{array}{llll} (1) & \mathsf{a} \to \mathsf{f}(\mathsf{c}), & (2) & \mathsf{a} \to \mathsf{h}(\mathsf{c}) \\ (3) & \mathsf{f}(x) \to \mathsf{h}(\mathsf{g}(x)), & (4) & \mathsf{h}(x) \to \mathsf{f}(\mathsf{g}(x)) \end{array} \right\}.$$

Let $s = \mathsf{f}(\mathsf{c})$ and $t = \mathsf{h}(\mathsf{c})$. We have $\mathcal{U}_{\mathsf{r}}(\mathcal{R}, s) \cup \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t) = \{(3), (4)\}$. Take an $\mathcal{F}$-algebra $\mathcal{A} = \langle \mathbb{N}, \langle f^{\mathcal{A}} \rangle_{f \in \mathcal{F}} \rangle$ as $\mathsf{a}^{\mathcal{A}} = \mathsf{c}^{\mathcal{A}} = 0$, $\mathsf{g}^{\mathcal{A}}(n) = n + 1$, $\mathsf{f}^{\mathcal{A}}(n) = n$, $\mathsf{h}^{\mathcal{A}}(n) = n + 1$. Then $[\![\mathsf{f}(x)]\!]_\sigma - [\![\mathsf{h}(\mathsf{g}(x))]\!]_\sigma = \sigma(x) - (\sigma(x) + 2) = -2$ and $[\![\mathsf{h}(x)]\!]_\sigma - [\![\mathsf{f}(\mathsf{g}(x))]\!]_\sigma = (\sigma(x) + 1) - (\sigma(x) + 1) = 0$. Take $k = 2$. Then $[\![\mathsf{f}(x)]\!]_\sigma \equiv [\![\mathsf{h}(\mathsf{g}(x))]\!]_\sigma \pmod{k}$ and $[\![\mathsf{h}(x)]\!]_\sigma \equiv [\![\mathsf{f}(\mathsf{g}(x))]\!]_\sigma \pmod{k}$ for any valuation σ. Furthermore, since we have $[\![s]\!] = [\![\mathsf{f}(\mathsf{c})]\!] = 0$ and $[\![t]\!] = [\![\mathsf{h}(\mathsf{c})]\!] = 1$, $[\![s]\!] \not\equiv [\![t]\!] \pmod{k}$. Hence, $\mathrm{NJ}(s,t)$ by Corollary 7.

5 Proving Non-joinability by Ordering

In Corollary 7, we considered the case that the carrier set is a set of integers. In such a case, another obvious choice to obtain a partition of the carrier set is to divide it as $A = \{n \in A \mid n < k\} \uplus \{n \in A \mid k \leq n\}$ for some k. We first formulate this idea in a more abstract setting, using the notion of ordered $\mathcal{F}$-algebra [35].

An *ordered $\mathcal{F}$-algebra* $\mathcal{A} = \langle A, \leq, \langle f^{\mathcal{A}} \rangle_{f \in \mathcal{F}} \rangle$ is a triple of a set A, a partial order $\leq$ on it and a tuple of functions $f^{\mathcal{A}} : A^n \to A$ for each n-ary function symbol $f \in \mathcal{F}$. We use $<$ to denote strict part of $\leq$, i.e. $< \,=\, \leq \setminus \geq$. An ordered $\mathcal{F}$-algebra $\mathcal{A} = \langle A, \leq, \langle f^{\mathcal{A}} \rangle_{f \in \mathcal{F}} \rangle$ is said to be *weakly monotone* if $a \leq b$ implies $f^{\mathcal{A}}(\ldots, a, \ldots) \leq f^{\mathcal{A}}(\ldots, b, \ldots)$ for any $a, b \in A$ and $f \in \mathcal{F}$. Interpretations of terms on ordered $\mathcal{F}$-algebras are defined in the same way as on $\mathcal{F}$-algebras.

Theorem 10. *Let $\mathcal{A}$ be a weakly monotone ordered $\mathcal{F}$-algebra and s, t be ground terms. Suppose (i) $[\![l]\!]_\sigma \leq [\![r]\!]_\sigma$ for any valuation σ and any $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)$, (ii) $[\![l]\!]_\sigma \geq [\![r]\!]_\sigma$ for any valuation σ and any $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$ and (iii) $[\![s]\!] > [\![t]\!]$. Then* $\mathrm{NJ}(s, t)$.

Proof. By weak monotonicity, for any valuation σ, $u \to_{\mathcal{U}_{\mathsf{r}}(\mathcal{R},s)} v$ implies $[\![u]\!]_\sigma \leq [\![v]\!]_\sigma$ and $u \to_{\mathcal{U}_{\mathsf{r}}(\mathcal{R},t)} v$ implies $[\![u]\!]_\sigma \geq [\![v]\!]_\sigma$. Hence the claim follows. □

Remark that well-foundedness of the ordering is not necessary, in contrast to orderings used in termination proving.

We now consider the case that term algebras are taken as $\mathcal{F}$-algebras, and formulate the theorem in a more general way using the notion of rewrite relation. For this, the following notion is useful.

Definition 11 (discrimination pair). *A pair $\langle \gtrsim, \succ \rangle$ of two relations $\gtrsim$ and $\succ$ is said to be a* discrimination pair *if (i) $\gtrsim$ is a rewrite relation, (ii) $\succ$ is a irreflexive relation and (iii) $\gtrsim \circ \succ \subseteq \succ$ and $\succ \circ \gtrsim \subseteq \succ$.*

Remark that neither transitivity, well-foundedness nor closure under substitutions and contexts is needed for the relation $\succ$, unlike a similar notion used in the termination proving, called reduction pair [24]. Note also that in the condition (iii), both of $\gtrsim \circ \succ \subseteq \succ$ and $\succ \circ \gtrsim \subseteq \succ$ are requested—this is again contrasted with the reduction pair where either $\gtrsim \circ \succ \subseteq \succ$ or $\succ \circ \gtrsim \subseteq \succ$ suffices; both conditions will be required in the proof of the theorem given below.

Clearly, for any rewrite quasi-order $\gtrsim$, the pair $\langle \gtrsim, \gtrsim \setminus \lesssim \rangle$ forms a discrimination pair.

Theorem 12. *Let $\mathcal{R}$ be a TRS and s, t ground terms. Suppose there exists a discrimination pair $\langle \gtrsim, \succ \rangle$ such that $\mathcal{U}_{\mathsf{r}}(\mathcal{R}, s) \subseteq \lesssim$, $\mathcal{U}_{\mathsf{r}}(\mathcal{R}, t) \subseteq \gtrsim$ and $s \succ t$. Then* $\mathrm{NJ}(s, t)$.

Proof. Since $\gtrsim$ is a rewrite relation, it follows that $u \to_{\{l \to r\}} v$ implies $u \lesssim v$ for any $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)$, and $u \to_{\{l \to r\}} v$ implies $u \gtrsim v$ for any $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$. Suppose $u \in [s](\xrightarrow{*}) \cap [t](\xrightarrow{*})$. Let $s = s_0 \to s_1 \to \cdots \to s_n = u$. Then, by Lemma 4, $s = s_0 \to_{l_{i_1} \to r_{i_1}} s_1 \to_{l_{i_2} \to r_{i_2}} \cdots \to_{l_{i_n} \to r_{i_n}} s_n = u$ with $l_{i_j} \to r_{i_j} \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)$ for all $j = 1, \ldots, n$. Thus $s \lesssim \cdots \lesssim u$. Since $t \prec s \lesssim \cdots \lesssim u$, we obtain $t \prec u$ by the property $\gtrsim \circ \succ \subseteq \succ$ of the discrimination pair. Similarly, from $t \to \cdots \to u$, we obtain $t \gtrsim \cdots \gtrsim u$. By $u \succ t \gtrsim \cdots \gtrsim u$, we obtain $u \succ u$ by the property $\succ \circ \gtrsim \subseteq \succ$ of the discrimination pair. This contradicts our assumption that $\succ$ is irreflexive. □

In terms of interpretations, Theorem 12 amounts to taking term algebras as $\mathcal{F}$-algebras, while Theorem 10 allows to take any $\mathcal{F}$-algebra. On the other hand, in terms of discrimination pairs, Theorem 10 amounts to taking a discrimination pair of the form $\langle \gtrsim, \gtrsim \setminus \lesssim \rangle$. Hence Theorem 10 is not subsumed by Theorem 12 and vice versa.

6 Argument Filtering for Non-joinability

The criterion of Theorem 12 has a typical style used in criteria for termination proving. Therefore, similarly to the termination proving case, a discrimination pair can be obtained using various path orders combined with argument filtering. For dependency pairs, usable rules can be considered after performing argument filtering [14] and this sometimes decreases usable rules that need to be considered. In this section, we show that such an extension is possible also for non-joinability proving.

An *argument filtering* [4] is a mapping $\pi : \mathcal{F} \to (\mathrm{List}(\mathbb{N}^+) \cup \mathbb{N}^+)$ such that $\pi(f) \in \{[i_1, \ldots, i_k] \mid 1 \le i_1 < \cdots < i_k \le \mathrm{arity}(f)\} \cup \{i \mid 1 \le i \le \mathrm{arity}(f)\}$. Here, $\mathbb{N}^+$ denotes the set of positive integers and $\mathrm{List}(\mathbb{N}^+)$ the set of lists of positive integers. The application of the argument filtering π to terms is recursively defined as $x^\pi = x$ for $x \in \mathcal{V}$, $f(t_1, \ldots, t_n)^\pi = f(t_{i_1}^\pi, \ldots, t_{i_k}^\pi)$ if $\pi(f) = [i_1, \ldots, i_k]$, $f(t_1, \ldots, t_n)^\pi = t_i^\pi$ if $\pi(f) = i$. Hence $t^\pi \in \mathrm{T}(\mathcal{F}^\pi, \mathcal{V})$ where $\mathcal{F}^\pi = \{f \in \mathcal{F} \mid \pi(f) \in \mathrm{List}(\mathbb{N}^+)\}$ with $\mathrm{arity}(f) = |\pi(f)|$. For a TRS $\mathcal{R}$, we put $\mathcal{R}^\pi = \{l^\pi \to r^\pi \mid l \to r \in \mathcal{R}\}$. For substitution θ, we put $\theta^\pi(x) = \theta(x)^\pi$.

The following properties of the argument filtering are well-known (e.g. [4]).

Lemma 13. *(1)* $(s\theta)^\pi = s^\pi\theta^\pi$*. (2)* $s \to_{\{l\to r\}} t$ *implies* $s^\pi \overset{=}{\to}_{\{l^\pi \to r^\pi\}} t^\pi$*.*

Theorem 14. *Let* $\mathcal{R}$ *be a TRS and* s, t *ground terms. Suppose there exist a discrimination pair* $\langle \gtrsim, \succ \rangle$ *and an argument filtering* π *such that* $\mathcal{U}_{\mathsf{r}}(\mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)^\pi, s^\pi) \subseteq {\lesssim}$*,* $\mathcal{U}_{\mathsf{r}}(\mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)^\pi, t^\pi) \subseteq {\gtrsim}$ *and* $s^\pi \succ t^\pi$*. Then* $\mathrm{NJ}(s, t)$*.*

Proof. Suppose $u \in [s](\overset{*}{\to}) \cap [t](\overset{*}{\to})$. Let $s = s_0 \to_{\{l_1 \to r_1\}} \cdots \to_{\{l_n \to r_n\}} s_n = u$. Then by Lemma 4, $l_i \to r_i \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)$ for all $i = 1, \ldots, n$. Then $s^\pi = s_0^\pi \overset{=}{\to}_{\{l_1^\pi \to r_1^\pi\}} s_1^\pi \overset{=}{\to}_{\{l_2^\pi \to r_2^\pi\}} \cdots \overset{=}{\to}_{\{l_n^\pi \to r_n^\pi\}} s_n^\pi = u^\pi$ by Lemma 13 where $l_i^\pi \to r_i^\pi \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)^\pi$ for all $i = 1, \ldots, n$. Then by Lemma 4, $l_{i_j}^\pi \to r_{i_j}^\pi \in \mathcal{U}_{\mathsf{r}}(\mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)^\pi, s^\pi)$. Hence $t^\pi \prec s^\pi \lesssim s_1^\pi \lesssim \cdots \lesssim u^\pi$. Thus, by the definition of the discrimination pair, $t^\pi \prec u^\pi$. Similarly, we have $t^\pi \gtrsim \cdots \gtrsim u^\pi$. Hence $u^\pi \succ t^\pi \gtrsim \cdots \gtrsim u^\pi$, and $u^\pi \succ u^\pi$. This contradicts $\succ$ is irreflexive. □

If one takes an argument filtering π such that $\pi(f) = [1, 2, \ldots, \mathrm{arity}(f)]$ for all $f \in \mathcal{F}$, then we have $\pi(t) = t$ for any term t. Thus, Theorem 14 subsumes Theorem 12.

Some readers may wonder whether Theorem 14 can be obtained from Theorem 12 by taking the discrimination pair $\langle \gtrsim', \succ' \rangle$ defined by $s \gtrsim' t$ iff $s^\pi \gtrsim t^\pi$ and $s \succ' t$ iff $s^\pi \succ t^\pi$ for some discrimination pair $\langle \gtrsim, \succ \rangle$. Indeed, it can be shown that the discrimination pair $\langle \gtrsim', \succ' \rangle$ given like this is again a discrimination pair. However, the direct application of Theorem 12 only yields $\mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$ in the place of $\mathcal{U}_{\mathsf{r}}(\mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)^\pi, t^\pi)$ in Theorem 14. Since the inclusion $\mathcal{U}_{\mathsf{r}}(\mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)^\pi, t^\pi) \subseteq \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$ may be proper, Theorem 14 is not subsumed by Theorem 12.

Example 15. Let

$$\mathcal{R} = \left\{ \begin{array}{llll} (1) & \mathsf{c} \to \mathsf{f}(\mathsf{c}, \mathsf{d}), & (2) & \mathsf{c} \to \mathsf{h}(\mathsf{c}, \mathsf{d}) \\ (3) & \mathsf{f}(x, y) \to \mathsf{h}(\mathsf{g}(y), x), & (4) & \mathsf{h}(x, y) \to \mathsf{f}(\mathsf{g}(y), x) \end{array} \right\}.$$

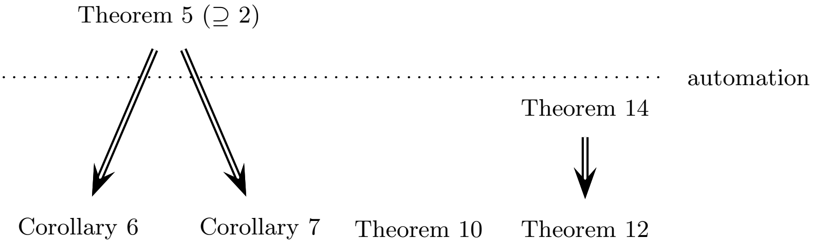


Fig. 1. Relations of theorems and corollaries

Let $s = \mathsf{h}(\mathsf{f}(\mathsf{c},\mathsf{d}),\mathsf{d})$ and $t = \mathsf{f}(\mathsf{c},\mathsf{d})$. First consider to apply Theorem 12. Then we need to solve the following constraint:

$$\left\{\begin{array}{llllll} \mathsf{h}(\mathsf{f}(\mathsf{c},\mathsf{d}),\mathsf{d}) & \succ \mathsf{f}(\mathsf{c},\mathsf{d}), & \mathsf{c} & \simeq \mathsf{f}(\mathsf{c},\mathsf{d}), & \mathsf{c} \simeq \mathsf{h}(\mathsf{c},\mathsf{d}) \\ \mathsf{f}(x,y) & \simeq \mathsf{h}(\mathsf{g}(y),x), & \mathsf{h}(x,y) & \simeq \mathsf{f}(\mathsf{g}(y),x) & \end{array}\right\}.$$

This constraint can not be satisfied using a discrimination pair $\langle \gtrsim_{rpo}, \gtrsim_{rpo} \setminus \lesssim_{rpo} \rangle$ based on recursive path orders. Next, we consider applying Theorem 14. For this, take an argument filtering π as $\pi(\mathsf{g}) = 1$, $\pi(\mathsf{f}) = [2]$ and $\pi(\mathsf{h}) = [1]$. Then we have $\mathcal{U}_{\mathsf{r}}(\mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)^\pi, s^\pi) = \{(3)^\pi, (4)^\pi\}$ and $\mathcal{U}_{\mathsf{r}}(\mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)^\pi, t^\pi) = \{(3)^\pi, (4)^\pi\}$. Then we need to solve the following constraint:

$$\left\{ \mathsf{h}(\mathsf{f}(\mathsf{d})) \succ \mathsf{f}(\mathsf{d}), \quad \mathsf{f}(y) \simeq \mathsf{h}(y), \quad \mathsf{h}(x) \simeq \mathsf{f}(x) \right\}.$$

Then the constraint is satisfied by a discrimination pair $\langle \gtrsim_{rpo}, \gtrsim_{rpo} \setminus \lesssim_{rpo} \rangle$, where $\gtrsim_{rpo}$ is the recursive path order based on the precedence $\mathsf{f} \simeq \mathsf{h}$. Thus $\mathrm{NJ}(s,t)$ by Theorem 14.

In Figure 1, we summarize relations of theorems and corollaries presented in the paper. The dotted line at the middle of the figure indicates that criteria below this line are suitable for automation.

7 Related Works

Non-Joinability Check in Confluence Provers We now review methods for proving non-joinability employed in the state-of-the-art confluence provers ACP [3], CSI [33] and Saigawa [17] that participated in the 1st Confluence Competition (CoCo 2012). Obviously, if the termination proof of the input TRS succeeds, the well-known criterion that confluence coincides with joinability of all critical pairs (Knuth-Bendix criterion [22]) can be used to prove non-confluence. Below we describe other approaches employed by these provers for proving non-confluence and non-joinability.

ACP basically uses the following three conditions to show non-joinability.

ACP(1). Fix some $n > 0$. Check $[s](\xrightarrow{*}) = [s](\rightarrow^{\leq n})$ and $[t](\xrightarrow{*}) = [t](\rightarrow^{\leq n})$. If it is the case and $[s](\rightarrow^{\leq n}) \cap [t](\rightarrow^{\leq n}) = \emptyset$ then conclude $\mathrm{NJ}(s,t)$.

ACP(2). Check $s(\epsilon) \neq t(\epsilon)$. If it is the case, check (by an approximation) that $\forall s' \in [s](\xrightarrow{*}).\ s(\epsilon) = s'(\epsilon)$ and $\forall t' \in [t](\xrightarrow{*}).\ t(\epsilon) = t'(\epsilon)$ hold. If they hold, conclude $\mathrm{NJ}(s,t)$.

ACP(3). If $s(\epsilon) = t(\epsilon) \notin \{l(\epsilon) \mid l \to r \in \mathcal{R}\}$, then check $\mathrm{NJ}(s|_i, t|_i)$ holds for some i. If it is the case, conclude $\mathrm{NJ}(s,t)$. This is used in conjunction with ACP(1) and ACP(2).

CSI uses the following two conditions to show non-joinability [33].

CSI(1). If $\mathrm{TCAP}(s)$ and $\mathrm{TCAP}(t)$ are not unifiable, then conclude $\mathrm{NJ}(s,t)$.

CSI(2). Use the approximation technique based on tree automata: Try to construct tree automata $\mathcal{A}_s$ and $\mathcal{A}_t$ such that $[s](\xrightarrow{*}) \subseteq \mathcal{L}(\mathcal{A}_s)$ and $[t](\xrightarrow{*}) \subseteq \mathcal{L}(\mathcal{A}_t)$ (using the method in [23]). If they succeed, then check $\mathcal{L}(\mathcal{A}_s) \cap \mathcal{L}(\mathcal{A}_t) = \emptyset$. If it is the case, then conclude $\mathrm{NJ}(s,t)$.

Saigawa uses CSI(1) above and the following extension of the Knuth-Bendix criterion [21] to disprove confluence.

Saigawa(1). Suppose $\mathcal{S}$ is confluent, $\mathcal{R}$ is terminating relative to $\mathcal{S}$, and $\mathcal{R}$ is not strongly overlapping on $\mathcal{R}$ and vice versa. Then $\mathcal{R} \cup \mathcal{S}$ is confluent iff all $\mathcal{S}$-critical pairs of $\mathcal{R}$ is joinable by $\mathcal{R} \cup \mathcal{S}$-rewrite steps.

Apparently the approach presented in the paper is very different from those employed already in these confluence provers, and the criteria given in the paper are not subsumed by any of the techniques employed in these confluence provers[3].

Decidable Classes. Besides Knuth-Bendix criterion saying that confluence is decidable for terminating TRSs [22], decision procedures for deciding confluence of the TRSs in some classes of TRSs have been investigated.

One of the most basic such classes is the class of ground TRSs [8,26]. For ground TRSs a polynomial time algorithm for deciding confluence is known [6,10]. Such a decision procedure is implemented in CSI [33].

Other well-known such classes include the class of shallow right-linear TRSs [15], the class of right-ground TRSs [16,20] and the class of monadic right-linear TRSs [28]. (The former two classes include the class of ground TRSs.) To the best of our knowledge, however, no implementations of the decision procedures other than the one mentioned above have been reported.

The criteria presented in this paper are free of syntactic restrictions on rewrite rules characterizing such classes. In particular, Examples 8, 9 and 15 are neither shallow, right-ground nor monadic; hence, they do not belong to any classes mentioned above for which confluence is decidable.

[3] Meanwhile, new versions of tools have been released, and CoCo 2013 have been held. The methods described in the present paper have been incorporated to the latest version of ACP.

8 Implementations and Experiments

Implementations The following instances of presented criteria have been implemented. The implementation is built on ACP. The language used in the implementation is the functional programming language SML/NJ [29].

Cor. 7 ($k = 2, 3$). Corollary 7 applied for the polynomial interpretation with linear polynomials, i.e. $f^{\mathcal{A}}$ has the form $a_{0,f} + a_{f,1}x_1 + \cdots + a_{n,f}x_n$ for each function symbol f of arity n. In case $k = 2$, we check whether $[\![l]\!]_\sigma - [\![r]\!]_\sigma$ is even for all rewrite rules $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s) \cup \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$ and whether $[\![s]\!] - [\![t]\!]$ is odd. We encode these constraints in boolean formulas and check the constraints by an external SAT solver (or SMT solver). We deal with integer variables of the range between 0 and 15 that are encoded by four bits. Thus the constraint that an integer variable $x = (b_3\ b_2\ b_1\ b_0)_{10}$ is even is encoded by $\bar{b}_0$ (i.e. b_0 equals false). The condition that a monomial ax ($a \in \mathbb{Z}$) is even is encoded by true if a is even and by "x is even" otherwise. The condition that a polynomial $a_0 + a_1x_1 + \cdots + a_nx_n$ is even is encoded recursively by the disjunct of both of the monomial a_0 and the polynomial $a_1x_1 + \cdots + a_nx_n$ are even and both are odd, recursively. Finally, the condition that meta polynomial $\Phi = \varphi_o + \varphi_1X_1 + \cdots + \varphi_nX_n$ where φ_i are polynomials, is even is encoded by all polynomials $\varphi_o, \ldots, \varphi_n$ are even and Φ is odd is encoded by the constant part φ_0 is odd and polynomials $\varphi_1, \ldots, \varphi_n$ are even. The case $k = 3$ is more complicated but encoding is again straightforward. For example, $(b_3\ b_2\ b_1\ b_0)_{10} \equiv (c_3\ c_2\ c_1\ c_0)_{10} \pmod 3$ can be encoded by $(b_3 \wedge b_2 \wedge b_1 \wedge b_0) \vee ((b_3 \otimes b_1) \wedge (b_2 \otimes b_0)) \vee (\bar{b_3} \wedge \bar{b_2} \wedge \bar{b_1} \wedge \bar{b_0})$.

Th. 10 (poly). Theorem 10 applied for polynomial interpretation with linear polynomials. Similar to the case Cor. 7 ($k = 2, 3$), we encode the constraints in boolean formulas and check the constraints by an external SAT solver. We also deal with integer variables of the range between 0 and 15 which are encoded by four bits. To ensure the weak monotonicity, we restrict all coefficients of $f^{\mathcal{A}} = a_{f,0} + a_{f,1}x_1 + \cdots + a_{f,n}x_n$ to be non-negative. Our implementation tries two possible applications of the Theorem to show $\mathrm{NJ}(s, t)$, namely that (1) $[\![s]\!] > [\![t]\!]$, $[\![l]\!]_\sigma \geq [\![r]\!]_\sigma$ for $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$ and $[\![l]\!]_\sigma \leq [\![r]\!]_\sigma$ for $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)$, and (2) $[\![t]\!] > [\![s]\!]$, $[\![l]\!]_\sigma \geq [\![r]\!]_\sigma$ for $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)$ and $[\![l]\!]_\sigma \leq [\![r]\!]_\sigma$ for $l \to r \in \mathcal{U}_{\mathsf{r}}(\mathcal{R}, t)$. Because of our bounds on integer variables, it may be the case that only one of (1) or (2) works and the other doesn't.

Th. 14 (rpo). Theorem 14 applied for recursive path order with argument filtering. Similar to the cases Cor. 7 ($k = 2, 3$) and Th. 10 (poly), we encode the constraints in boolean formulas and check the constraints by an external SAT solver. Let $\mathcal{S} = \mathcal{U}_{\mathsf{r}}(\mathcal{R}, s)$. We approximate the set of usable rules $\mathcal{U}_{\mathsf{r}}(\mathcal{S}^\pi, s^\pi)$ by $\hat{\mathcal{U}_{\mathsf{r}}}(\mathcal{S}^\pi, s^\pi)$ where $\hat{\mathcal{U}_{\mathsf{r}}}$ is given by $\hat{\mathcal{U}_{\mathsf{r}}}(\mathcal{R}', s') = \mathcal{U}(\mathcal{R}', s') \cup \{l \to r \in \mathcal{R}' \mid l \in \mathcal{V}\}$ and $\mathcal{U}$ returns the set of usable rules for dependency pairs [4]. The soundness of the approximation follows from $\mathcal{U}_{\mathsf{r}}(\mathcal{R}', s') \cap \gtrsim\ \subseteq \hat{\mathcal{U}_{\mathsf{r}}}(\mathcal{R}', s') \cap \gtrsim$ for well-founded rewrite quasi-order $\gtrsim$. Here, $\mathcal{S}$ is computed before the encoding and the constraint $\hat{\mathcal{U}_{\mathsf{r}}}(\mathcal{S}^\pi, s^\pi) \subseteq\ \gtrsim$ is encoded in a similar way as the encoding of termination criteria using dependency pairs [14]. Currently, we don't know whether a direct encoding of $\mathcal{U}_{\mathsf{r}}(\mathcal{S}^{\pi}, s^{\pi})$ is possible. Finally, as in the case for Th. 10 (poly), our implementation tries two possible applications of the Theorem (the $s^{\pi} \succ t^{\pi}$ version and the $t^{\pi} \succ s^{\pi}$ version).

Selecting candidates for the non-joinability test For all these implementations, candidates for the non-joinability test are generated from the input TRS $\mathcal{R}$ like this: (1) first compute the one-step unfolding $\mathcal{R}'$ of $\mathcal{R}$ [27] and then (2) compute critical pairs of $\mathcal{R} \cup \mathcal{R}'$, and finally, (3) all critical pairs are sorted w.r.t. term size and at most 100 critical pairs are considered as candidates.

Apparently, various ways to compute candidates for the non-joinability test are possible. Note that considering just reducts of critical pairs of $\mathcal{R}$ for candidates is not enough for proving non-confluence [11].

Here we explain very roughly which candidates for the non-joinability test are considered in the state-of-the-art confluence provers. According to [33], CSI considers candidates from the set $\mathcal{C} = \bigcup_{\langle s_1, t_1 \rangle} \{\langle s, t \rangle \mid s_1 \to^{\leq m} s, t_1 \to^{\leq n} t\}$ where $\langle s_1, t_1 \rangle$ ranges over those satisfying $s_1 = C[r_1\sigma]_p \leftarrow C[l_1\sigma]_p = C[l_2\sigma]_q \to C[r_2\sigma]_q = t_1$ with $l_1 \to r_1, l_2 \to r_2 \in \mathcal{R}$, $p \leqslant q$ and $p \in \mathrm{Pos}(C[l_2]_q)$. ACP uses similar candidates, with (probably very) different heuristics for choosing C, p, q, m, n and choices of the candidates from $\mathcal{C}$. Saigawa considers, for testing CSI(1), candidates from the set $\bigcup_{\langle -, v, - \rangle} \{\langle s, t \rangle \mid v \to^{\leq n} s, v \to^{\leq n} t, s \neq t\}$ where $\langle -, v, - \rangle$ ranges over critical peaks of $\mathcal{R} \cup \mathcal{R}^{-1}$.

Experiments Experiments have been performed on our implementation and the state-of-the-art confluence provers ACP (ver. 0.31), CSI (ver. 0.2) and Saigawa (ver. 1.4). Each test is performed on a PC with one 2.50GHz CPU and 4G memory; the timeout is set to 60 seconds. We have tested a collection of 23 new examples which includes Examples 8, 9, 15 developed in the course of experiments, and a collection of 35 examples from the 1st Confluence Competition (CoCo 2012) that were not proved to be confluent by any of participating provers.

A summary of the experiments is shown in Table 1. Each column shows success(✓) or failure(×) of confluence disproving on Examples 8, 9 and 15, the numbers of examples from the collections that are successfully proved to be non-confluent and of those that timeout (except CSI, for which one can not distinguish timeout and failure), and the total time in seconds. The column below all shows the result for the combination of the four instances. Note that ACP, CSI and Saigawa consume considerable time for proving confluence while our implementation concentrates on disproving confluence.

All provers ACP, CSI and Saigawa fail on Examples 8, 9 and 15. Both of Cor. 7 ($k = 2$) and Cor. 7 ($k = 3$) succeed on Examples 8 and 9. Th. 10 (poly) succeeds on Example 8. Th. 14 (rpo) succeed on Examples 8 and 15. Hence, incomparability of Cor. 7 and Th. 14 (rpo) is observed.

In the experiments on the collection of 23 new examples, the following are observed: Th. 14 (rpo) succeeds most. Cor. 7 ($k = 2$) and Cor. 7 ($k = 3$) succeed on the same examples. The examples handled by Th. 10 (poly) are also handled by Th. 14 (rpo) and also by Cor. 7. Examples handled by any of the provers ACP, CSI and Saigawa are also handled by all.

Table 1. Summary of experiments

	ACP	CSI	Saigawa	Cor. 7	Cor. 7	Th. 10	Th. 14	all
				$(k=2)$	$(k=3)$	(poly)	(rpo)	
Example 8	×	×	×	✓	✓	✓	✓	✓
Example 9	×	×	×	✓	✓	×	×	✓
Example 15	×	×	×	×	×	×	✓	✓
23 examples (success)	9	12	3	16	16	14	19	21
23 examples (timeout)	0	–	1	0	3	0	0	1
23 examples (time in sec.)	2	2107	228	25	293	206	26	84
35 examples (success)	18	21	17	17	16	17	17	16
35 examples (timeout)	1	–	6	5	8	3	1	9
35 examples (time in sec.)	71	485	482	318	562	446	106	761

In the experiments on the collection of 35 examples from CoCo 2012, the following are observed. All instances succeed on the same examples, except for Cor. 7 $(k = 3)$, in which one timeouts. The numbers of examples on which ACP, CSI and Saigawa succeed but all fails are 4, 5, 3, respectively. There is one example (Cops Problem 15) which is proved by our implementation but by none of the provers. Unfortunately, the success on this example is not due to our new technique—this difference arises by the way one computes the candidates for non-joinability test.

Example 16 (Cops Problem 15). Let

$$\mathcal{R} = \left\{ \begin{array}{ll} (1) & \mathsf{f}(x, \mathsf{f}(y, z)) \to \mathsf{f}(\mathsf{f}(x, y), \mathsf{f}(x, z)) \\ (2) & \mathsf{f}(\mathsf{f}(x, y), z) \to \mathsf{f}(\mathsf{f}(x, z), \mathsf{f}(y, z)) \\ (3) & \mathsf{f}(\mathsf{f}(x, y), \mathsf{f}(y, z)) \to y \end{array} \right\}.$$

Let $s = \mathsf{f}(\mathsf{a}, \mathsf{a})$ and $t = \mathsf{a}$. Note $s, t \in [\mathsf{f}(\mathsf{f}(\mathsf{a}, \mathsf{a}), \mathsf{f}(\mathsf{a}, \mathsf{a}))](\xrightarrow{*})$. Then s, t are normal forms and hence it is easy to see that $\mathrm{NJ}(s, t)$.

Finally, the running time is observed like this: Th. 14 (rpo) < Cor. 7 $(k = 2)$ $\ll$ Th. 10 (poly) $\ll$ Cor. 7 $(k = 3)$.

All details of the experiments are available on the webpage: `http://www.nue.riec.tohoku.ac.jp/tools/acp/experiments/frocos13/all.html`.

9 Conclusion

We have presented sufficient criteria of non-joinability of terms that can be used to disprove confluence of TRSs. Our criteria are based on interpretation and ordering, and are using new notions of usable rules and discrimination pairs. The combination of arguments filtering and our notion of usable rules have been also considered. We have given some concrete instances of our criteria which are amenable for automation—implementations of these instances have been described and experiments have been reported. Experiments have shown that the presented methods can automatically disprove confluence of TRSs, on which state-of-the-art automated confluence provers fail.

Since our criteria are parametrized by $\mathcal{F}$-algebras or orderings, other concrete instances of our criteria can be possibly used. We note that all of our instances are highly non-optimal, i.e. they do not use the full strength of discrimination pairs; for example, they are all well-founded although this is not required for discrimination pairs. Future work would involve exploring other possibilities to obtain effective interpretations and orderings.

Acknowledgements. Thanks are due to anonymous referees for many valuable comments. This work was partially supported by a grant from JSPS No. 23500002.

References

1. Aoto, T.: Automated confluence proof by decreasing diagrams based on rule-labelling. In: Proc. of 21st RTA. LIPIcs, vol. 6, pp. 7–16. Schloss Dagstuhl (2010)
2. Aoto, T., Toyama, Y.: A reduction-preserving completion for proving confluence of non-terminating term rewriting systems. Logical Methods in Computer Science 1(31), 1–29 (2012)
3. Aoto, T., Yoshida, J., Toyama, Y.: Proving confluence of term rewriting systems automatically. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 93–102. Springer, Heidelberg (2009)
4. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236(1-2), 133–178 (2000)
5. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
6. Comon, H., Godoy, G., Nieuwenhuis, R., Tiwari, A.: The confluence of ground term rewrite systems is decidable in polynomial time. In: Proc. of 42nd LICS, pp. 263–297. IEEE Computer Society Press (2001)
7. Contejean, E., Marché, C., Tomás, A.P., Urbain, X.: Mechanically proving termination using polynomial interpretation. Journal of Automated Reasoning 34, 325–363 (2005)
8. Dauchet, M., Heuillard, T., Lescanne, P., Tison, S.: Decidability of the confluence of finite ground term rewrite systems and of other related term rewrite systems. Information and Computation 88, 187–201 (1990)
9. Durand, I., Middeldorp, A.: Decidable call by need computations in term rewriting. In: McCune, W. (ed.) CADE 1997. LNCS (LNAI), vol. 1249, pp. 4–18. Springer, Heidelberg (1997)
10. Felgenhauer, B.: Deciding confluence of ground term rewrite systems in cubic time. In: Proc. of 23rd RTA. LIPIcs, vol. 15, pp. 165–175. Schloss Dagstuhl (2012)
11. Felgenhauer, B.: A proof order for decreasing diagrams. In: Proc. of 1st IWC, pp. 9–15 (2012)
12. Genet, T.: Decidable approximations of sets of descendants and sets of normal forms. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 151–165. Springer, Heidelberg (1998)
13. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 216–231. Springer, Heidelberg (2005)

14. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Mechanizing and improving dependency pairs. Journal of Automated Reasoning 37(3), 155–203 (2006)
15. Godoy, G., Tiwari, A.: Confluence of shallow right-linear rewrite systems. In: Ong, L. (ed.) CSL 2005. LNCS, vol. 3634, pp. 541–556. Springer, Heidelberg (2005)
16. Godoy, G., Tiwari, A., Verma, R.: Characterizing confluence by rewrite closure and right ground term rewriting systems. Applicable Algebra in Engineering, Communication and Computing 15, 13–36 (2004)
17. Hirokawa, N., Klein, D.: Saigawa: A confluence tool. In: Proc. of 1st IWC, p. 49 (2012)
18. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool: Techniques and features. Information and Computation 205(4), 474–511 (2007)
19. Hirokawa, N., Middeldorp, A.: Decreasing diagrams and relative termination. Journal of Automated Reasoning 47(4), 481–501 (2011)
20. Kaiser, Ł.: Confluence of right ground term rewriting systems is decidable. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 470–489. Springer, Heidelberg (2005)
21. Klein, D., Hirokawa, N.: Confluence of non-left-linear TRSs via relative termination. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 258–273. Springer, Heidelberg (2012)
22. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970)
23. Korp, M., Middeldorp, A.: Match-bounds revisited. Information and Computation 207(11), 1259–1283 (2009)
24. Kusakari, K., Nakamura, M., Toyama, Y.: Argument filtering transformation. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 47–61. Springer, Heidelberg (1999)
25. Middeldorp, A.: Approximating dependency graphs using tree automata techniques. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 593–610. Springer, Heidelberg (2001)
26. Oyamaguchi, M.: The Church-Rosser property for ground term rewriting systems is decidable. Theoretical Computer Science 49, 43–79 (1987)
27. Payet, É.: Loop detection in term rewriting using eliminating unfoldings. Theoretical Computer Science 403, 307–327 (2008)
28. Salomaa, K.: Decidability of confluence and termination of monadic term rewriting systems. In: Book, R.V. (ed.) RTA 1991. LNCS, vol. 488, pp. 275–286. Springer, Heidelberg (1991)
29. Standard ML of New Jersey, `http://www.sml.org/`
30. Terese: Term Rewriting Systems. Cambridge University Press (2003)
31. Toyama, Y.: Confluent term rewriting systems. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, p. 1. Springer, Heidelberg (2005) slides are available from, `http://www.nue.riec.tohoku.ac.jp/user/toyama/slides/toyama-RTA05.pdf`
32. Urbain, X.: Modular & incremental automated termination proofs. Journal of Automated Reasoning 32, 315–355 (2004)
33. Zankl, H., Felgenhauer, B., Middeldorp, A.: CSI – A confluence tool. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 499–505. Springer, Heidelberg (2011)
34. Zankl, H., Felgenhauer, B., Middeldorp, A.: Labelings for decreasing diagrams. In: Proc. of 22nd RTA. LIPIcs, vol. 10, pp. 377–392. Schloss Dagstuhl (2011)
35. Zantema, H.: Termination of term rewriting by semantic labelling. Fundamenta Informaticae 24, 89–105 (1995)

On Forward Closure and the Finite Variant Property*

Christopher Bouchard[1], Kimberly A. Gero[1],
Christopher Lynch[2], and Paliath Narendran[1]

[1] University at Albany—SUNY, Albany, NY, USA
{cbou,kgero001,dran}@cs.albany.edu
[2] Clarkson University, Potsdam, NY, USA
clynch@clarkson.edu

Abstract. Equational unification is an important research area with many applications, such as cryptographic protocol analysis. Unification modulo a convergent term rewrite system is undecidable, even with just a single rule. To identify decidable (and tractable) cases, two paradigms have been developed — Basic Syntactic Mutation [14] and the Finite Variant Property [6]. Inspired by the Basic Syntactic Mutation approach, we investigate the notion of forward closure along with suitable redundancy constraints. We show that a convergent term rewriting system R has a finite forward closure if and only if R has the finite variant property. We also show the undecidability of the finiteness of forward closure, therefore determining if a system has the finite variant property is undecidable.

Keywords: Equational unification, Finite variant property, Forward closure, Term rewriting, Undecidability.

1 Introduction

Equational unification is an important research area which has applications in cryptographic protocol analysis, automated theorem proving, and automated reasoning. However, unification modulo a convergent term rewrite system is undecidable in general, even if the system has just a single rule [1]. Consequently, there is interest in identifying decidable instances of equational unification. Two important syntactic paradigms have been developed to identify such instances. One paradigm was developed in "Basic Syntactic Mutation", by Christopher Lynch and Barbara Morawska [14]. They give syntactic criteria on equational axioms E which guarantee that the corresponding E-unification problem is in **NP**. If the system satisfies some additional criteria, they provide a polynomial-time decision algorithm for that E-unification problem. The second paradigm was developed in "The finite variant property: How to get rid of some algebraic

* C. Bouchard, K. Gero, and P. Narendran were supported in part by NSF grant CNS 09-05286. C. Lynch was supported in part by NSF grant CNS 09-05378.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 327–342, 2013.

properties" by Hubert Comon-Lundh and Stéphanie Delaune [6]. Here it was shown that E-unification is decidable if E has the finite variant property, and Escobar, Meseguer, and Sasse showed how narrowing can be used to implement an E-unification decision algorithm for such an E [9].

In studying the BSM algorithm in the context of convergent rewrite systems [5], we found that the notion of saturation by paramodulation is equivalent to that of forward closure if the system is convergent and suitable redundancy constraints are added. Hermann considers the idea of forward closure chains in "Chain Properties of Rule Closures" [10], and he proved that the finiteness of forward closure is undecidable for general rewrite systems—in particular, the system he considers has an undecidable termination problem. Hermann did not, however, consider any sort of redundancy.

In this paper, we extend the notion forward closure[1] to allow redundancy constraints and show that a convergent term rewriting system R has a finite forward closure if and only if R has the finite variant property. In showing this equivalence we define the IR-boundedness property which characterizes the finite variant property. Additionally, we show the undecidability of the finiteness of forward closure for *convergent* rewrite systems, and therefore that determining if a system has the finite variant property for such systems is undecidable. Finally, we show that the finiteness of forward closure is a modular property, i.e., if two disjoint rewrite systems have a finite forward closure, their union also has a finite forward closure.

In the interest of space, several proofs and examples have been omitted or shortened in this version. They are given in full in a tech report [4].

2 Notation and Preliminaries

We consider rewrite systems over ranked signatures, usually denoted Σ, and a possibly infinite set of variables, usually denoted $\mathcal{X}$. We assume the reader is familiar with the usual notions and concepts in term rewriting systems [2] and equational unification [3]. The set of all terms over Σ and $\mathcal{X}$ is denoted as $T(\Sigma, \mathcal{X})$. Given a term t, we denote by $\mathcal{P}os(t)$ the set of all positions in t, and by $\mathcal{FP}os(t)$ the set of all non-variable positions in t. An *equation*, e.g. in [2] is an ordered pair of terms (s, t), usually written as $s \approx t$. Here s is the *left-hand side* and t is the *right-hand side* of the equation [2]. A *rewrite rule* is an equation $s \approx t$ where $\mathcal{V}ar(s) \supseteq \mathcal{V}ar(t)$, usually written as $s \to t$. A *term rewriting system* is a set of rewrite rules.

Our focus in this paper is on unifiability modulo theories that have convergent term rewriting systems. Let R be a convergent term rewriting system. We assume that there is a well-founded reduction ordering $\succ$ on terms such that $\to_R^+ \subseteq \succ$. Let $\prec$ be the inverse of $\succ$, i.e., $s \prec t$ if and only if $t \succ s$. We further assume that the ordering is total on ground terms. We extend this order to equations as $(s \approx t) \succ (u \approx v)$ if and only if $\{s, t\} \succ_{mul} \{u, v\}$, where $\succ_{mul}$ is the multiset

[1] From this point on, we will use "forward closure" to mean "forward closure *with redundancy constraints*".

order induced by $\succ$. A term t is an *innermost redex* of a rewrite system R if and only if all proper subterms of t are irreducible and t is an instance of the left-hand side of a rule in R.

The following proposition holds since $\rightarrow_R \subseteq \succ$ and since $\succ$ is transitive.

Proposition 1. *Let R be a convergent rewrite system and let t, l, and r be terms such that $t \succ l$ and $t \succ r$. If $l \downarrow_R r$, then every term that appears in the rewrite proof ("valley proof") is below t in the reduction ordering $\succ$.*

3 Strict Redundancy

Given a set of equations E, the set of ground instances of equations in E is denoted by $Gr(E)$. An instance is ground if its terms do not contain any variables. A ground equation e is *strictly redundant in E* if and only if it is a consequence of equations in $Gr(E)$ which are smaller than e modulo the ordering we use to show termination [14]. An equation e is *strictly redundant in E* if and only if every ground instance e' of e is strictly redundant in E. In our setting, with convergent rewriting systems R and reduction orderings $\succ$, this can be formulated as follows. For a ground equation $s \approx t$ we define the following (possibly infinite) ground term rewriting system:

$$\mathcal{G}_R^{\prec(s\approx t)} := \{l \rightarrow r \mid (l \rightarrow r) \in Gr(R) \text{ and } (l \rightarrow r) \prec (s \approx t)\}$$

Now a ground equation $s \approx t$ is strictly redundant in R if and only if

$$\mathcal{G}_R^{\prec(s\approx t)} \vdash s \approx t$$

Since our focus in this paper is on convergent rewrite systems, we first give a condition on R such that $\mathcal{G}_R^{\prec(s\approx t)}$ is convergent.

Lemma 1. *Let R be a convergent rewrite system, and let s and t be ground terms such that $s \succ t$. Then $\mathcal{G}_R^{\prec(s\approx t)}$ is convergent.*

Now we explore conditions on equations that force those equations to be redundant in a rewrite system. The following lemma follows almost directly from the definition of $\mathcal{G}_R^{\prec(s\approx t)}$.

Lemma 2. *Let R be a convergent rewrite system. Then an equation $s_1 \approx s_2$ is strictly redundant in R if and only if for every ground instance $\delta(s_1) \approx \delta(s_2)$ of $s_1 \approx s_2$, $\delta(s_1)$ and $\delta(s_2)$ are joinable modulo $\mathcal{G}_R^{\prec(\delta(s_1)\approx\delta(s_2))}$.*

Lemma 3. *Suppose R is a convergent rewrite system such that the rule $l \rightarrow r$ is strictly redundant in R. Then the rule $\theta(l) \rightarrow \theta(r)$ is strictly redundant in R for any substitution θ.*

Lemma 4. *Let R be a convergent rewrite system, and let l and r be terms joinable modulo R such that $l \succ r$ and a proper subterm of l is reducible. Then $l \approx r$ is strictly redundant in R.*

Proof. Suppose $l \approx r$ is not strictly redundant in R. Then, by Lemma 2, there is a ground instance $\delta(l) \approx \delta(r)$ such that $\delta(l)$ and $\delta(r)$ are not joinable in $\mathcal{G} = \mathcal{G}_R^{\prec(\delta(l)\approx\delta(r))}$. Since a proper subterm of l is reducible, there is a rule $l' \to r'$ in R and a position $p \neq \epsilon$ in $\mathcal{P}os(l)$ such that $l|_p = \sigma(l')$ and $l \to_R l[\sigma(r')]_p$. Therefore $\delta(l)|_p = \delta(\sigma(l'))$ and $\delta(l) \to_R \delta(l[\sigma(r')]_p)$. Since $\to_R \subseteq \succ$, we have that $\delta(l) \succ \delta(l[\sigma(r')]_p)$, and since reduction orders are closed under substitutions, $\delta(l) \succ \delta(r)$. Thus, by Proposition 1, there are rewrite sequences

$$\delta(r) \to_R^* t \leftarrow_R^* \delta(l[\sigma(r')]_p)$$

such that each term in the rewrite sequences is below $\delta(l)$ in the ordering $\prec$. Therefore, since each term is ground, $\delta(r)$ and $\delta(l[\sigma(r')]_p)$ are joinable in $\mathcal{G}$.

Since $\delta(l) = \delta(l[\sigma(l')]_p) = \delta(l)[\delta(\sigma(l'))]_p$, and since the ordering $\prec$ has the subterm property on ground terms, $\delta(\sigma(l)) \prec \delta(l)$. Thus $\delta(l) \to_{\mathcal{G}} \delta(l[\sigma(r')]_p)$. So $\delta(l)$ and $\delta(r)$ are joinable in $\mathcal{G}$, which is a contradiction. Therefore $s \approx r$ is strictly redundant in R. □

Lemma 5. *Let R be a convergent rewrite system and $l \approx r$ be an equation such that l is an innermost redex and $l \to_R^+ r$. Then $l \approx r$ is strictly redundant in R if there is a term r' such that $l \to_R r'$ and $r' \prec r$.*

Proof. If $l \to_R r'$ and l is an innermost redex, then $l \to r'$ is an instance of a rule in R and $(l \approx r') \prec (l \approx r)$. Since R is confluent, $r' \downarrow_R r$ and, by Proposition 1, every term that appears in the rewrite proof is below l in the ordering. Thus every ground instance of $l \to r$ can be proven using only smaller instances of rules in R, and therefore $l \approx r$ is strictly redundant in R. □

Unfortunately, the converse cannot be proved unless additional assumptions are made about the ordering. However, for ground equations we can prove both directions:

Lemma 6. *Let R be a convergent rewrite system and $l \approx r$ be a* ground *equation such that l is an innermost redex and $l \to_R^+ r$. Then $l \approx r$ is strictly redundant in R if and only if there is a ground term r' such that $l \to_R r'$ and $r' \prec r$.*

Proof. The "if" part follows from Lemma 5.

Suppose now that there is no term $r' \prec r$ such that $l \to_R r'$, but $l \approx r$ is strictly redundant in R. Then by Lemma 2, l and r must be joinable modulo $\mathcal{G}_R^{\prec(l\approx r)}$. Thus there must be a rule $l \to r''$ in $\mathcal{G}_R^{\prec(l\approx r)}$, and so $(l \to r'') \prec (l \approx r)$. We then have that $r'' \prec r$. This is a contradiction, so $l \approx r$ is not strictly redundant in R. □

This leads us to a very useful lemma. In practice many of the equations we look at will be rewrite rules whose right-hand side is in normal form. This gives us a simple syntactic check for the redundancy of such rules.

Lemma 7. *Let R be a convergent rewrite system, and let $l \approx r$ be an equation such that l is reducible and r is the normal form of l. Then $l \approx r$ is strictly redundant in R if and only if a proper subterm of l is reducible.*

4 A (Slightly) Stronger Notion of Redundancy

A rule $\rho_1 = l_1 \rightarrow r_1$ is said to be an instance of a rule $\rho_2 = l_2 \rightarrow r_2$ if and only if there is a substitution σ such that $\sigma(l_2) = l_1$ and $\sigma(r_2) = r_1$. We write this as $\rho_2 \sqsupseteq \rho_1$ or as $\rho_2 \sqsupseteq_\sigma \rho_1$ if the substitution σ is of significance. For instance, the rule $f(x, x) \rightarrow x$ is an instance of the rule $f(x, y) \rightarrow x$.

A rule ρ is *redundant*[2] in R if and only if it is either strictly redundant in R (i.e., every ground instance of ρ is strictly redundant in R) or there is a rule ρ' in R such that $\rho' \sqsupseteq \rho$.

We can extend Lemma 3 from the previous section to redundancy as follows.

Lemma 8. *Let R be a convergent rewrite system such that the rule $l \rightarrow r$ is redundant in R. Then the rule $\theta(l) \rightarrow \theta(r)$ is redundant in R for any substitution θ.*

5 Forward Closure

Following Hermann [10], the *forward-closure* of a term rewrite system R is defined in terms of the following operation on rules in R. Let $\rho_1 = l_1 \rightarrow r_1$ and $\rho_2 = l_2 \rightarrow r_2$ be two rules in R, and let $p \in \mathcal{FP}os(r_1)$. Then

$$\rho_1 \rightsquigarrow_p \rho_2 \ := \ \sigma(l_1 \rightarrow r_1[r_2]_p)$$

where $\sigma = mgu(r_1|_p =^? l_2)$. We call this the *forward overlap* of ρ_1 and ρ_2 at p.

Proposition 2. *Let ρ_1, ρ_2, and ρ_3 be rules such that $\rho_3 = \rho_1 \rightsquigarrow_p \rho_2$ for some position p. If $t \rightarrow_{\rho_3} t'$ then $\exists\, t''\colon t \rightarrow_{\rho_1} t''$ and $t'' \rightarrow_{\rho_2} t'$.*

Given rewrite systems R_1, R_2, and R_3 we define $\mathcal{FOV}(R_1, R_2)$ (the set of forward overlaps) and $\mathcal{N}(R_1, R_2, R_3)$ (the set of non-redundant rules) as

$$\mathcal{FOV}(R_1, R_2) := \{\rho_1 \rightsquigarrow_p \rho_2 \mid \rho_1 = (l_1 \rightarrow r_1) \in R_1,\ \rho_2 \in R_2, \text{ and } p \in \mathcal{FP}os(r_1)\}$$
$$\mathcal{N}(R_1, R_2, R_3) := \{\rho \mid \rho \in \mathcal{FOV}(R_1, R_2) \text{ and } \rho \text{ is not redundant in } R_3\}$$

We now simultaneously define $NR_k(R)$ (new rules step) and $FC_k(R)$ (forward closure step) for all $k \geq 0$.

$$\begin{aligned} NR_0(R) &:= R & NR_{k+1}(R) &:= \mathcal{N}(NR_k(R), R, FC_k(R)) \\ FC_0(R) &:= R & FC_{k+1}(R) &:= FC_k(R) \ \cup \ NR_{k+1}(R) \end{aligned}$$

Finally, we define the forward closure of R.

$$FC(R) := \bigcup_{i=1}^{\infty} FC_i(R)$$

Note that $FC_k(R) \subseteq FC_{k+1}(R)$ for all $k \geq 0$. A set of rewrite rules R is *forward-closed* if and only if $FC(R) = R$.

[2] This is referred to as *non-strictly redundant* in [15].

Example 1. The following rewrite system has a finite forward closure:

$$R_{\text{ex}} = \{f(s(x)) \to f(x),\ s(s(s(x))) \to x\}$$

There is an overlap of the first rule with itself, and we see that the rewrite system has one forward overlap,

$$\mathcal{FOV}(NR_0(R_{\text{ex}}),\, R_{\text{ex}}) = \{f(s(s(x))) \to f(x)\}$$

This rule is not redundant in R_{ex}, as the ground instance $f(s(s(a))) \approx f(a)$ cannot be proven by $\mathcal{G}_{R_{\text{ex}}}^{\prec(f(s(s(a)))\approx f(a))}$, i.e. smaller rules in $Gr(R_{\text{ex}})$. Thus we see that

$$NR_1(R_{\text{ex}}) = \{f(s(s(x))) \to f(x)\}$$
$$FC_1(R_{\text{ex}}) = \{f(s(s(x))) \to f(x),\ f(s(x)) \to f(x),\ s(s(s(x))) \to x\}$$

To compute the next set of forward overlaps, we can only overlap the new rule with the first rule of R_{ex}. So there is one new forward overlap,

$$\mathcal{FOV}(NR_1(R_{\text{ex}}),\, R_{\text{ex}}) = \{f(s(s(s(x)))) \to f(x)\}$$

However, this rule is redundant by Lemma 7, since the subterm $s(s(s(x)))$ at position 1 of the left-hand side is reducible. Thus $NR_2(R_{\text{ex}}) = \emptyset$, and the rewrite system has a finite forward closure $FC(R_{\text{ex}}) = FC_1(R_{\text{ex}})$. □

Now we will give constraints that must be satisfied to have a finite forward closure.

Lemma 9. *Given a convergent rewrite system R, $FC(R)$ is finite if and only if there is a $k > 0$ such that $NR_k(R) = \emptyset$.*

Corollary 1. *Given a convergent rewrite system R, $FC(R)$ is finite if and only if there is a $k > 0$ such that $FC(R) = FC_k(R)$.*

Now we will discuss the case where a term t is an innermost redex.

Lemma 10. *Let R be a convergent rewrite system, and let t and t' be terms where t is an innermost redex. If $t \to_{FC_{k'}(R)} t'$ then $t \to_R^k t'$ for some $k \leq k'+1$.*

Proof. Suppose $k' = 0$. Then $FC_{k'}(R) = R$, and thus $t \to_R t'$.

Otherwise, assume that if $t \to_{FC_{k'-1}(R)} t'$ then $t \to_R^k t'$ for some $k \leq k'$. If $t \to_{FC_{k'}(R)} t'$ then either $t \to_{FC_{k'-1}(R)} t'$ or $t \to_{NR_{k'}(R)} t'$. In the first case we are done. In the second case, $t \to t'$ is in $NR_{k'}(R) = \mathcal{N}(NR_{k'-1}(R), R, FC_{k'-1}(R))$. Therefore $(t \to t') = \rho_1 \rightsquigarrow_p \rho_2$, for ρ_1 in $NR_{k'-1}(R)$, ρ_2 in R, and position p. Since $NR_{k'-1}(R) \subseteq FC_{k'-1}(R)$, $t \to_{FC_{k'-1}(R)} t'' \to_R t'$ for some t''. By our assumption, $t \to_R^k t''$ for some $k \leq k'$, so $t \to_R^{k+1} t'$. □

In the next lemma we show that when our initial rewrite system R is convergent then at every step in our forward closure procedure the rewrite system returned is convergent.

Lemma 11. *Let R be a convergent rewrite system. Then for all $k \geq 0$, $FC_k(R)$ is convergent.*

Throughout the remainder of the section we will show that our forward closure procedure will get an innermost redex "closer and closer" to its normal form. The section culminates in a theorem that will be used to show one of the main results in this paper.

Lemma 12. *Let R be a convergent rewrite system, and let t and t' be terms where t is a* ground *innermost redex and $t \to_{FC_k(R)} t'$ for some $k \geq 0$. If t' is not in normal form then there exists a term $t'' \prec t'$ such that $t \to_{FC_{k+1}(R)} t''$.*

Proof (Sketch). If t' is not in normal form, then there is some rule in $FC_k(R)$ that rewrites t to t'. This rule will be overlapped with a rule from R in the next step of forward closure, resulting in a new rule to a lower term. □

Lemma 13. *Suppose R is a convergent rewrite system and t an innermost redex with normal form $\hat{t}$ where $t \to_R^{k'} \hat{t}$. Then there is a k such that $t \to_{FC_k(R)} \hat{t}$.*

Proof. Let θ be a substitution that maps each variable x in t to a distinct free constant c_x. Let $s = \theta(t)$ and $\hat{s} = \theta(\hat{t})$. Note that $\theta(\hat{t})$ is still irreducible, so $\hat{s}$ is the normal form of s. Also note that, by Lemma 11, since R is convergent so is $FC_k(R)$ for any $k \geq 0$.

Suppose there is no k such that $s \to_{FC_k(R)} \hat{s}$. Then, by Lemma 12, if $s \to_{FC_k(R)} s_k$ for some k and some ground term s_k, then there is a ground term $s_{k+1} \prec s_k$ such that $s \to_{FC_{k+1}(R)} s_{k+1}$. Thus there is an infinitely descending chain

$$s \succ \cdots \succ s_k \succ s_{k+1} \succ s_{k+2} \succ \cdots$$

and therefore the ordering $\succ$ is not well-founded. This is a contradiction, so there must be a k such that $s \to_{FC_k(R)} \hat{s}$. Since $s = \theta(t)$ is an innermost redex, this rewrite occurs at the root. Thus there is a rule $\rho = (l \to r)$ in $FC_k(R)$ such that $\rho \sqsupseteq_\sigma (\theta(t) \to \theta(\hat{t}))$.

Suppose now that t does not rewrite to its normal form in one step modulo $FC_k(R)$. Then $\rho \not\sqsupseteq (t \to \hat{t})$. If $\theta \sqsupseteq_\tau \sigma$, then $\rho \sqsupseteq_\tau (\theta(t) \to \theta(\hat{t}))$ since $\theta \circ \theta = \theta$ (i.e., θ is idempotent). But then $\rho \sqsupseteq_\sigma (t \to \hat{t})$. So $\theta \not\sqsupseteq \sigma$. This means there is a position p in l such that $l|_p = c_x$ for some x. This is a contradiction since each c_x is free. Thus $t \to_{FC_k(R)} \hat{t}$. □

Corollary 2. *If R is a convergent rewrite system and t an innermost redex with normal form $\hat{t}$, then $t \to_{FC(R)} \hat{t}$.*

Theorem 1. *A convergent rewrite system R is forward-closed if and only if every innermost redex can be reduced to its R-normal form in one step.*

Proof. If $R = FC(R)$ then, by Corollary 2, for any innermost redex t with normal form $\hat{t}$, $t \to_R \hat{t}$. Thus we have proven the "only if" part.

To prove the "if" part, assume that every innermost redex can be reduced to its normal form in one step, but R is not forward-closed. Thus there is a rule

$l \to r$ in $FC(R)$ that is not in R. If l is not an innermost redex in R then, by Lemma 4, $l \to r$ is redundant in R. So l must be an innermost redex in R and can be reduced to its normal form $\hat{l}$ in one step. Since $(l \to \hat{l}) \prec (l \to r)$, and since R is confluent, l and r are joinable using only smaller instances of rules in R and thus $l \to r$ is redundant in R. This is a contradiction, so R must be forward-closed. □

6 Equivalence of Finiteness of Forward Closure and the Finite Variant Property

In this section we show that a system has a finite forward closure (with redundancy) if and only if it has the *finite variant property*, as defined by Comon-Lundh and Delaune [6]. We will adopt the notation used in [7].

Definition 1. *Let R be a convergent rewrite system. A term-substitution pair (t, θ) is an R-*variant *of a term s if and only if θ is R-normalized and $\theta(s) \to^{!}_{R} t$. An R-variant (t, θ) of a term s is said to be more general than another R-variant (t', θ') of the same term s, denoted as $(t, \theta) \sqsupseteq (t', \theta')$, if and only if there is a substitution ρ such that $t' = \rho(t)$ and $\theta' = \rho \circ \theta$. A complete set of R-variants of a term s, denoted as $[\![s]\!]^{\star}$, is a set of R-variants of s, such that for every R-variant (s', γ) of s there is a variant $(t, \theta) \in [\![s]\!]^{\star}$ such that $(t, \theta) \sqsupseteq (s', \gamma)$. A convergent term rewriting system R has the* finite variant property *if and only if every term s has a* finite *complete set of R-variants.*

Comon-Lundh and Delaune showed that the finite variant property is equivalent to the *boundedness* property.

Definition 2. *A rewrite system R has the boundedness property (or is bounded) if, for every term t, there exists an integer n such that for every normalized substitution σ, the normal form of $\sigma(t)$ is reachable by a derivation whose length can be bounded by n (thus independently of σ):*

$$\forall t\, \exists n\, \forall \sigma \colon (\sigma{\downarrow})(t) \xrightarrow{\leq n}_{R} \sigma(t){\downarrow}$$

We first introduce a different notion of boundedness for a term rewriting system and prove that this new notion is equivalent to the standard notion.

Definition 3. *A rewrite relation $\to_R$ (alternatively, a term rewriting system R) is* IR-bounded *if and only if there is a "global" bound n such that every innermost redex can be reduced to its normal form in n steps or less:*

$$\exists n\, \forall t \colon \left[t \text{ is an innermost redex} \Rightarrow t \xrightarrow{\leq n}_{R} t{\downarrow} \right]$$

Lemma 14. *Suppose a convergent rewrite system R is bounded. Then R is IR-bounded.*

Proof. For each function symbol f in Σ, consider the term $t_f = f(x_1, \ldots, x_m)$, where m is the arity of f and $x_1, \ldots, x_m$ are variables. Since R is bounded, there is an n_f such that for any normalized substitution θ, $\theta(t_f) \xrightarrow{\leq n_f}_R \theta(t_f){\downarrow}$. Let u be a innermost redex with f as its root symbol. Note that there is a normalized substitution θ such that $\theta(t_f) = u$, and thus $u \xrightarrow{\leq n_f}_R u{\downarrow}$. Let n be the largest such n_f for any f in Σ. Then for any innermost redex u', $u' \xrightarrow{\leq n}_R u'{\downarrow}$. Therefore, R is IR-bounded. □

Lemma 15. *Suppose a convergent rewrite system R is IR-bounded. Then R is bounded.*

Proof. Since R is IR-bounded, there is a bound n such that for any innermost redex u, $u \xrightarrow{\leq n}_R u{\downarrow}$. Let t be a term, and θ be a normalized substitution. The set of positions where $\theta(t)$ could be rewritten is a subset of $\mathcal{FP}os(t)$. Consider a position p in $\mathcal{FP}os(t)$ such that $\theta(t)|_p$ is an innermost redex. Since R is IR-bounded, $\theta(t)|_p \xrightarrow{\leq n}_R (\theta(t)|_p){\downarrow}$. Once $\theta(t)|_p$ is rewritten, the only subterms that can become new innermost redexes are its ancestors. Clearly then the entire term $\theta(t)$ can be rewritten in no more than $n \cdot |\mathcal{FP}os(t)|$ steps. Therefore R is bounded. □

With this result, we can easily show one direction of the equivalence.

Lemma 16. *Suppose a convergent rewrite system R has a finite forward closure $FC(R)$. Then R has the finite variant property.*

Proof. If $FC(R)$ is finite, then $FC(R) = FC_k(R)$ for some k. By Corollary 2, given an innermost redex t, $t \to_{FC(R)} t{\downarrow}$. So $t \to_{FC_k(R)} t{\downarrow}$, and by Lemma 10 there is a $k' \leq k + 1$ such that $t \to_R^{k'} t{\downarrow}$. Therefore R is IR-bounded. By Lemma 15, R is bounded, and thus R has the finite variant property. □

In the other direction, things are a bit more complicated. We relate the variants of a rewrite system to redundancy. First, given a rewrite system R, we define the following set of rules, $\mathcal{V}_R$.

Definition 4. *For a convergent rewrite system R that has the finite variant property, we define*

$$\mathcal{V}_R = \{\theta(l) \to l' \mid l \to r \in R \textit{ and } (l', \theta) \in [\![l]\!]^\star \textit{ and } \theta(l) \textit{ is an innermost redex}\}$$

The rules in $\mathcal{V}_R$ correspond to variants of the left-hand sides of rules in R. The next three lemmas use this set to prove that a convergent system with the finite variant property has a finite forward closure.

Lemma 17. *Suppose a convergent rewrite system R has the finite variant property. Then there is a $k > 0$ such that each rule in $\mathcal{V}_R$ is redundant in $FC_k(R)$.*

Proof. Since R has the finite variant property, for any term t, $[\![l]\!]^\star$ is finite. Thus $\mathcal{V}_R$ is finite. For each $\theta(l) \to l'$ in $\mathcal{V}_R$, $\theta(l)$ is an innermost redex and l' is its normal form. Thus, by Lemma 13, there is a $k > 0$ such that $\theta(l) \to_{FC_k(R)} l'$. Let k' be the max of all such k. Each rule in $\mathcal{V}_R$ is redundant in $FC_{k'}(R)$. □

Lemma 18. *Suppose a convergent rewrite system R has the finite variant property, and let $k > 0$ be such that each rule in $\mathcal{V}_R$ is redundant in $FC_k(R)$. Then every innermost redex can be reduced to its normal form in one step modulo $FC_k(R)$.*

Proof. Let $\theta(l)$ be an innermost redex where l is the left-hand side of a rule in R. Let s be its normal form. Clearly the substitution θ has to be a normalized substitution (over $\mathcal{V}ar(l)$) for otherwise $\theta(l)$ would not be an innermost redex. Since R has the finite variant property, there is a variant (l', σ) of l such that $(s, \theta) \sqsubseteq (l', \sigma)$. Thus there is a substitution η such that $\theta = \eta \circ \sigma$ and $s = \eta(l')$. Thus, since $\sigma(l)$ is also an innermost redex, $\theta(l) \to s$ is an instance of the rule $\sigma(l) \to l' \in \mathcal{V}_R$. Since l' is the normal form of $\sigma(l)$, by Lemma 7, $\sigma(l) \to l'$ must not be strictly redundant in $FC_k(R)$. So $\sigma(l) \to l'$, and therefore $\theta(l) \to s$, must be an instance of a rule in $FC_k(R)$ and we are done. □

Lemma 19. *Suppose a convergent rewrite system R has the finite variant property. Then R has a finite forward closure $FC(R)$.*

We have now equated the finite variant property to the finiteness of forward closure. All the results in this section lead us to the following theorem.

Theorem 2. *Let R be a convergent rewrite system. The following statements are equivalent:*

(i) R is bounded.
(ii) R is IR-bounded.
(iii) R has a finite forward closure
(iv) R has the finite variant property

7 Undecidability of Finiteness of Forward Closure

We will prove the undecidability of the finiteness of forward closure by reduction from the uniform mortality problem for deterministic Turing machines [11]. Given a deterministic Turing machine M, the machine is said to be *uniformly mortal* if and only if there is a number k such that, for any instantaneous description I of M, the number of transitions that M can make starting from I is at most k.

We represent a deterministic Turing machine M as a tuple $(\Gamma, \flat, Q, \delta, F)$, where Γ is the tape alphabet, $\flat \in \Gamma$ is the blank symbol, Q is the set of states, $F \subset Q$ is the set of final states, and $\delta\colon (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function. We assume that $\Gamma \cap Q = \emptyset$.

An *instantaneous description* (ID) of M is represented as a tuple (u, q, γ, v), where u is a suffix of the string to the left of the tape head, q is the current state,

γ is the current symbol under the tape head, and v is a prefix of the string to the right of the head. The strings to the left and right of the tape head may be infinite, but only a finite suffix and prefix, respectively, will contain non-blank symbols. Therefore, we let u be the longest suffix of the string to the left of the tape head such that $u \neq \not{b}u'$. Similarly, v is the longest prefix of the string to the right of the head such that $v \neq v'\not{b}$.

For IDs I_1 and I_2 of M, $I_1 \vdash I_2$ if and only if there is a transition in δ that would move M from I_1 to I_2. Note that this usage of $\vdash$ is separate from the usual meaning of "proves". An ID $I = (u, q, \gamma, v)$ is *final* if and only if $q \in F$.

The notion of an ID can be extended to that of a *window*. A window W of M is a tuple (u, q, γ, v) such that $u \in \not{b}^*u'$ and $v \in v'\not{b}^*$ for some u' and v' such that $I = (u', q, \gamma, v')$ is an ID of M. In this case, W *extends* I. The width of W is $|W| = |u| + |v| + 1$. For windows W_1 and W_2, $W_1 \vdash W_2$ if and only if $|W_1| = |W_2|$ and there are IDs I_1 and I_2 such that W_1 and W_2 extend I_1 and I_2, respectively, and $I_1 \vdash I_2$.

Proposition 3. *Let M be a Turing machine, and let I_1, I_2, ..., I_n be IDs of M such that $I_1 \vdash I_2 \vdash \cdots \vdash I_n$. Then there is a width k and windows W_1, W_2, ..., W_n, each with width k, such that each W_i extends I_i and $W_1 \vdash W_2 \vdash \cdots \vdash W_n$.*

For any given Turing machine M, we construct a rewrite system R_M and show that M is uniformly mortal if and only if $FC(R_M)$ is finite. Our system is over the signature $\Sigma = Q \cup \Gamma \cup \{\epsilon, s\}$, where each $q \in Q$ has arity 3, each $\gamma \in \Gamma$ has arity 1, ϵ is a constant, and s has arity 1. We assume an infinite set $\mathcal{X}$ of variables.

We can encode a number n as a term $s^n(\epsilon)$. Each ternary function symbol $q \in Q$ represents a window in state q, and each monadic function symbol $\gamma \in \Gamma$ represents concatenation on the left by that symbol. We encode a string $w = \gamma_1 \cdots \gamma_n$ over Γ as a term $enc(w) = (\gamma_1 \circ \cdots \circ \gamma_n)(\epsilon)$, where $\circ$ is function composition (i.e., $(f \circ g)(x) = f(g(x))$). We can then encode a window (u, q, γ, v) as a term $q(enc(u^{\text{rev}}), \gamma(enc(v)), s^n(\epsilon))$, where u^{rev} is the reverse of the string u, and n is the number of transitions the machine is allowed to make.

We say two terms t_1 and t_2 are *sequential* if and only if t_1 and t_2 both have root symbols from Q and $t_1|_3 = s(t_2|_3)$. We say a term t is *legal* if and only if there is a window W of M such that t encodes W. We say a term is *illegal* if and only if it has a root symbol from Q but is not legal.

Definition 5. *We define a function $\phi\colon T(\Sigma, \mathcal{X}) \to T(\Sigma)$ to transform illegal terms into legal terms. For all $q \in Q$,*

$$\phi(q(t_1, t_2, t_3)) = q(\phi'_\Gamma(t_1),\, \phi'_\Gamma(t_2),\, \phi'_{\{s\}}(t_3))$$

where $\phi'_S\colon T(\Sigma, \mathcal{X}) \to T(\Sigma)$ is a helper function parameterized by a signature $S \subseteq \Sigma$,

$$\phi'_S(t) = \begin{cases} f(\phi'_S(t')) & \textit{if } t = f(t') \textit{ for some } f^{(1)} \in S \\ \epsilon & \textit{otherwise} \end{cases}$$

The function ϕ'_S finds the "highest" occurrence of a term whose root symbol does not belong in a string over signature S and replaces it with ϵ. The function ϕ uses this to ensure that subterms encode valid tape strings (over the signature Γ) or numbers (over the signature $\{s\}$).

We can now construct our rewrite system R_M from a machine M.

Definition 6. *Let $M = (\Gamma, b\!\!\!/, Q, \delta, F)$ be a deterministic Turing machine. First set $R_M := \emptyset$. For each left-moving transition $(q, \gamma) \mapsto (q', \gamma', L)$ in δ, extend R_M by*

$$R_M := R_M \cup \{q(\gamma_0(x),\, \gamma(y),\, s(z)) \to q'(x,\, \gamma_0(\gamma'(y)),\, z) \mid \gamma_0 \in \Gamma\}$$

where x, y, and z are variables. Then, for each right-moving transition $(q, \gamma) \mapsto (q', \gamma', R)$ in δ, extend R_M by

$$R_M := R_M \cup \{q(x,\, \gamma(\gamma_0(y)),\, s(z)) \to q'(\gamma'(x),\, \gamma_0(y),\, z) \mid \gamma_0 \in \Gamma\}$$

where again, x, y, and z are variables.

We first prove some basic properties of the rewrite system R_M.

Lemma 20. *Let M be a deterministic Turing machine, let t_1 be an innermost redex, and let t_2 and t_3 be terms such that $t_1 \to_{R_M} t_2$ and $t_1 \to_{R_M} t_3$. Then $t_2 = t_3$.*

Lemma 21. *Let M be a deterministic Turing machine. Then the rewrite system R_M is convergent.*

Lemma 22. *Let M be a deterministic Turing machine, let t_1 be an innermost redex, and let t_2 be a term such that $t_1 \to_{R_M} t_2$. Then t_2 is either an innermost redex or in normal form.*

Lemma 23. *Let M be a deterministic Turing machine, let t_1 be an innermost redex, and let t_2 be a term such that $t_1 \to_{R_M} t_2$. Then t_1 and t_2 are sequential.*

Our goal in this section is to show that the rewrite system R_M models computation of the machine M. Unfortunately, there are terms over Σ that are R_M-reducible but do not encode any window of M. With the ϕ function, we can map such illegal terms to a representative legal term. The following lemma shows that ϕ preserves the R_M-reducibility of the term, and thus we can focus our attention on legal terms.

Lemma 24. *Let M be a deterministic Turing machine, and let t_1 be an innermost redex and t_2 be a term such that t_1 and t_2 have root symbols from Q. Then $t_1 \to^k_{R_M} t_2$ for some $k > 0$ if and only if $\phi(t_1) \to^k_{R_M} \phi(t_2)$.*

Proof (Sketch). The idea is that ϕ'_Γ and $\phi'_{\{s\}}$ can be pushed below the subterms of instances of rules in R_M. So if $t_1 \to^k_{R_M} t_2$, then for any step $t \to t'$, there is a rule $l \to r$ in R_M such that $t \to t' \sqsubseteq_\sigma l \to r$. If we apply ϕ, the ϕ'_Γ and $\phi'_{\{s\}}$ will be pushed down into $\sigma(x)$ for each $x \in \mathcal{V}ar(l)$, and thus $\phi(t) \to \phi(t')$. Therefore we have $\phi(t_1) \to^k_{R_M} \phi(t_2)$.

Conversely, if $t_1 \not\to^k_{R_M} t_2$, then applying ϕ cannot fix things, because it only changes things below the rule. Therefore $\phi(t_1) \not\to^k_{R_M} \phi(t_2)$. □

Corollary 3. *Let M be a deterministic Turing machine, and let t be a term with a root symbol from Q such that no proper subterm of t is reducible. Then $\phi(t)$ is in R_M-normal form if and only if t is in R_M-normal form.*

Now we can relate transitions between windows of M to rewriting terms that encode them in R_M.

Lemma 25. *Let M be a deterministic Turing machine, let W_1 and W_2 be windows of M with equal width, and let t_1 and t_2 be sequential terms encoding W_1 and W_2, respectively. Then $W_1 \vdash W_2$ if and only if $t_1 \to_{R_M} t_2$.*

Proof (Sketch). Here the idea is that if t_1 and t_2 encode W_1 and W_2, respectively, and if there is a transition from W_1 to W_2, then it corresponds to a unique rule in R_M that rewrites t_1 to t_2. Similarly, if there is a rule that rewrites t_1 to t_2, it corresponds to a unique transition from W_1 to W_2. □

Lemma 26. *Let M be a deterministic Turing machine, let W be a window of M, and let t be a term encoding W. If W is final, then t is in normal form.*

Lemma 27. *Let M be a deterministic Turing machine. Then M is uniformly mortal if and only if the rewrite system R_M is IR-bounded.*

Proof (Sketch). We first show a one-to-one correspondence between windows of M and legal terms. Transitions between windows correspond to rewrites in R_M. If the machine is uniformly mortal, the bound corresponds to IR-boundedness. Otherwise there exists some unbounded rewrite sequence starting from an innermost redex. □

Theorem 3. *It is undecidable to check, given a finite convergent term rewriting system, whether it has a finite forward closure.*

Proof. By Lemma 27, we have reduced the uniform mortality problem for deterministic Turing machines to the IR-boundedness problem. Therefore, by Theorem 2, the uniform mortality problem can be reduced to checking if R has a finite forward closure. By Lemma 21, for any deterministic Turing machine M we know that R_M is convergent. Thus it is undecidable whether a finite convergent term rewriting system has a finite forward closure. □

Corollary 4. *It is undecidable to check, given a finite convergent term rewriting system, whether it has the finite variant property.*

8 Modularity of Forward Closure

In this section we examine how forward closure behaves when rewrite systems are combined. We first consider the modularity of the finiteness of forward closure, i.e., whether the property is preserved when combining systems over disjoint signatures.

Theorem 4. *Let R_1 and R_2 be finite rewrite systems over signatures Σ_1 and Σ_2 respectively. If $\Sigma_1 \cap \Sigma_2 = \emptyset$, then $FC(R_1 \cup R_2) = FC(R_1) \cup FC(R_2)$.*

Proof. Suppose $FC(R_1 \cup R_2) \supsetneq FC(R_1) \cup FC(R_2)$. Then there must be a k such that either a rule from $FC_k(R_1)$ was overlapped with a rule from R_2, or a rule from $FC_k(R_2)$ with a rule from R_1. We will assume the former without loss of generality. Thus there is a rule $l \to r$ in $FC(R_1 \cup R_2)$ such that

$$(l \to r) = (l_1 \to r_1) \rightsquigarrow_p (l_2 \to r_2)$$

where $p \in \mathcal{FP}os(r_1)$, $(l_1 \to r_1) \in FC_k(R_1)$, and $(l_2 \to r_2) \in R_2$. So then

$$(l \to r) = \theta(l_1) \to \theta(r_1[r_2]_p)$$

where $\theta = mgu(r_1|_p =^? l_2)$. However, since Σ_1 and Σ_2 are disjoint, and since p is a non-variable position in r_1, the terms $r_1|_p$ and l_2 are not unifiable due to function clash. This is a contradiction. Since $FC(R_1 \cup R_2) \supseteq FC(R_1) \cup FC(R_2)$, we have that $FC(R_1 \cup R_2) = FC(R_1) \cup FC(R_2)$. □

However, if the systems are allowed to share constants, then even if the systems have finite forward closures their union may not.

Example 2. Let $R_1 = \{f(a, h(x)) \to h(f(b, x))\}$, and let $R_2 = \{b \to a\}$, where a and b are constants. These systems are clearly convergent and forward-closed. However, consider their union,

$$R_1 \cup R_2 = \{f(a, h(x)) \to h(f(b, x)),\ b \to a\}$$

This system is convergent. However, it has an infinite forward closure, because for all $k > 0$:

$$NR_{2k}(R_1 \cup R_2) = \{f(a, h^{k+1}(x)) \to h^{k+1}(f(a, x))\}$$

This is obtained by overlapping the rule from $NR_{2k-2}(R_1 \cup R_2)$ first with the rule from R_2, then with the rule from R_1 (this is why the rules occur in every *other* step of forward closure). None of these rules are redundant, because they are not instances of existing rules and the ground instances obtained by applying the substitution $\{x \mapsto a\}$ cannot be proven by smaller instances of rules. Since $NR_k(R_1 \cup R_2) \neq \emptyset$ for any k, by Lemma 9, $FC(R_1 \cup R_2)$ is not finite. □

9 Relationship to Runtime Complexity

Inspired by a comment from one of our reviewers, we examined the relationship to the field of *runtime complexity*, as described in [12]. The notion of the runtime complexity of a rewrite system is similar to the IR-boundedness property. However, while runtime complexity gives a bound for all rewrite sequence from an innermost redex, IR-boundedness only guarantees that a rewrite sequence *exists* which is shorter than the bound. For this reason, a rewrite system with $O(1)$ runtime complexity is IR-bounded, but it seems that the inverse is not necessarily true.

Several tools exist for automatically checking the runtime complexity of a rewrite system, such as CaT[3] and TCT[4]. These tools can now be used to recognize a class of rewrite systems with the finite variant property.

10 Conclusion and Future Work

Inspired by Basic Syntactic Mutation [5, 14], we explored forward closure and its relation to the finite variant property [6]. We found that, with suitable redundancy constraints, the finiteness of forward closure is equivalent to the finite variant property. We also showed that finiteness of forward closure is undecidable, even for convergent rewrite systems.

A great deal of research has gone into finding ways to decide if a rewrite system has the finite variant property [8]. As we have shown the equivalence of the finite variant property and finiteness of forward closure, we have a convenient procedure for checking the finite variant property, much like Knuth-Bendix completion provides a procedure for deciding the word problem [13]. As the finiteness of forward closure is undecidable, the procedure may not terminate, but if the rewrite system has the finite variant property, the procedure will terminate in a finite number of steps.

Our future work centers around extending forward closure to work modulo equational theories. The most important is the theory of AC (associativity and commutativity), which has many practical applications, but we hope to consider a much more general class of theories. We will also examine in more detail how forward closure behaves when rewrite systems are combined that are not completely disjoint.

Acknowledgements. We wish to thank Serdar Erbatur, Ralf Sasse, and the reviewers for their detailed comments, style suggestions, and proof corrections which helped considerably to improve this paper.

References

[1] 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] `http://cl-informatik.uibk.ac.at/software/cat/`

[4] `http://cl-informatik.uibk.ac.at/software/tct/`

[2] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1999)

[3] Baader, F., Snyder, W.: Unification Theory. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 440–526. Elsevier Science Publishers BV (1999)

[4] Bouchard, C., Gero, K.A., Lynch, C., Narendran, P.: On Forward Closure and the Finite Variant Property. Technical report, Dept. of Computer Science, University at Albany—SUNY (July 2013)

[5] Bouchard, C., Gero, K.A., Narendran, P.: Some Notes on Basic Syntactic Mutation. In: Escobar, S., Korovin, K., Rybakov, V. (eds.) Proceedings 26th International Workshop on Unification, pp. 9–14 (2012)

[6] 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)

[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] Escobar, S., Meseguer, J., Sasse, R.: Effectively checking the finite variant property. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 79–93. Springer, Heidelberg (2008)

[9] Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. Journal of Logic and Algebraic Programming 81(7-8), 898–928 (2012); Rewriting Logic and its Applications

[10] Hermann, M.: Chain properties of rule closures. Formal Aspects of Computing 2(1), 207–225 (1990)

[11] Hillebrand, G.G., Kanellakis, P.C., Mairson, H.G., Vardi, M.Y.: Undecidable Boundedness Problems for Datalog Programs. Journal of Logic Programming 25(2), 163–190 (1995)

[12] Hirokawa, N., Moser, G.: Automated Complexity Analysis Based on the Dependency Pair Method. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 364–379. Springer, Heidelberg (2008)

[13] Knuth, D.E., Bendix, P.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970)

[14] Lynch, C., Morawska, B.: Basic Syntactic Mutation. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 471–485. Springer, Heidelberg (2002)

[15] Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 371–443. Elsevier, MIT Press (2001)

Term Rewriting with Logical Constraints*

Cynthia Kop[1] and Naoki Nishida[2]

[1] Department of Computer Science, University of Innsbruck
Technikerstraße 21a, 6020 Innsbruck, Austria
Cynthia.Kop@uibk.ac.at

[2] Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
nishida@is.nagoya-u.ac.jp

Abstract. In recent works on program analysis, transformations of various programming languages to term rewriting are used. In this setting, *constraints* appear naturally. Several definitions which combine rewriting with logical constraints, or with separate rules for integer functions, have been proposed. This paper seeks to unify and generalise these proposals.

Keywords: Term rewriting, Constraints, Integer rewriting.

1 Introduction

Given the prevalence of computer programs in modern society, an important role is reserved for program analysis. Such analysis could take the form of for instance *termination* ("will this program end eventually, regardless of user input?"), *productivity* ("will this program stay responsive during its run?") and *equivalence* ("will this optimised code return the same result as the original?").

In recent years, there have been several results which transform a real-world program analysis problem into a query on term rewriting systems (TRSs). Such transformations are used to analyse termination of small, constructed languages (e.g. [3]), but also real code, like Java Bytecode [13], Haskell [10], LLVM [5], or Prolog [15]. Similar transformations are used to analyse code equivalence in [4,9].

In these works, *constraints* arise naturally. Where traditional TRSs generally consider well-founded sets like the natural number, more dedicated techniques are necessary when dealing with for instance integers or floating point numbers. Unfortunately, standard techniques for analysing TRSs are not equipped to also handle constraints. While integers and constraints *can* be encoded in TRSs, the results are either hairy or infinite, and generally hard to handle.

For this reason, rewriting with native support for logical constraints over a model was proposed [9]. While the results from normal term rewriting do not immediately apply in this setting, the *ideas* extend easily, so dedicated results are derived without much effort. Thus, constrained TRSs give a useful abstraction layer for program analysis. Several alternative definitions of constrained rewriting, focused on *integer constraints*, have also been given, see e.g. [4,5,8].

* The research in this paper is supported by the Austrian Science Fund (FWF) international project I963 and the Japan Society for the Promotion of Science.

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 343–358, 2013.

Unfortunately, the various formalisms are incompatible; results from one style of constrained rewriting do not necessarily transfer to another. This is a shame, as e.g. the lemma generation method in [12] (there used to prove equivalence of C-functions) might otherwise be reused in termination proofs. Also, dependency pairs and graphs are introduced in each of [3,8,14]. Thus, a lot of time is spent on redoing the same work for slightly different settings. Moreover, there are things we cannot do easily with any of them, such as overflow-conscious analysis.

In this paper, we propose a new formalism which unifies existing definitions of constrained rewriting. This formalism seeks to be *general*: unlike most of its predecessors, we do not limit interest to the integers, since in the future we will likely want to analyse programs which involve for instance real numbers or bitvectors. Moreover, we do not restrict attention to one kind of analysis (e.g. only termination or function equivalence). This way, we may for instance use the same dependency pair framework both to analyse termination of Haskell, and as part of a proof that two Java programs produce the same result (as termination is an essential property in inductive equivalence proofs, see e.g. [4,12]).

Paper Setup. This paper is structured as follows. In Section 2 we consider some preliminaries: both mathematical notions and a definition of many-sorted term rewriting. In Sections 3 and 4 we introduce the *LCTRS* formalism, which is the main contribution of this work. In Section 5 we will study how LCTRSs relate to existing definitions. Finally, to demonstrate how existing analysis techniques extend, we will consider basic confluence and termination results in Section 6.

2 Preliminaries

2.1 Sets and Functions

We assume that the mathematical notion of a *set* is well-understood.

The *function space* from a set A to a set B, denoted $A \Longrightarrow B$, consists of all sets f of pairs $\langle a, b\rangle$ with $a \in A$ and $b \in B$, such that for all $a \in A$ there is a unique $b \in B$ with $\langle a, b\rangle \in f$. The $\Longrightarrow$ is considered right-associative, so $A \Longrightarrow B \Longrightarrow C$ is the function space $A \Longrightarrow (B \Longrightarrow C)$. We use functional notation: for $f \in A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow B$, $f(a_1, \dots, a_n)$ denotes the unique b with $\langle a_1, \langle a_2, \dots \langle a_n, b\rangle \dots\rangle \in f$. When dealing with constraints, we need a notion of truth. To this end, we will often use the set $\mathbb{B} = \{\top, \bot\}$ of *booleans.*

Example 1. We consider the set $\mathbb{N}$ of natural numbers and the set $\mathbb{R}$ of real numbers. An example element of the function space $\mathbb{R} \Longrightarrow \mathbb{R} \Longrightarrow \mathbb{N}$ is the function $\lambda x \in \mathbb{R}, y \in \mathbb{R}.\ abs(\lceil x + y\rceil - 9)$. Here, the λ notation denotes function construction. The comparison relation $>$ on natural numbers is an element of $\mathbb{N} \Longrightarrow \mathbb{N} \Longrightarrow \mathbb{B}$, and can also be denoted in extended form, $\lambda x \in \mathbb{N}, y \in \mathbb{N}.\ x > y$.

2.2 Many-Sorted Term Rewriting Systems

Next, we consider term rewriting. In constrained rewriting, *types* like integers and booleans appear naturally. Since we have, at present, little reason to introduce function types, let us consider *many-sorted TRSs*.

Sorts and Signature. We assume given a set $\mathcal{S}$ of *sorts* and a set $\mathcal{V}$ of *variables*. A *signature* Σ is a set of *function symbols* f, each equipped with a *sort declaration* of the form $[\iota_1 \times \ldots \times \iota_n] \Rightarrow \kappa$ with all ι_i and κ sorts. A *variable environment* is a set Γ of variable : sort pairs.

Terms. Fixing a signature Σ, a *term* is any expression s built from function symbols in Σ, variables, commas and parentheses, such that $\Gamma \vdash s : \iota$ can be derived for some environment Γ and sort ι, using the following inference rules:

$$\frac{}{\Gamma \cup \{x : \iota\} \vdash x : \iota} \qquad \frac{\Gamma \vdash s_1 : \iota_1 \quad \ldots \quad \Gamma \vdash s_n : \iota_n \quad f : [\iota_1 \times \ldots \times \iota_n] \Rightarrow \kappa \in \Sigma}{\Gamma \vdash f(s_1, \ldots, s_n) : \kappa}$$

For any non-variable term s, there is a unique sort ι such that $\Gamma \vdash s : \iota$ for some Γ; we say that ι is the sort of s. The set of terms over Σ and $\mathcal{V}$ is denoted $\mathcal{T}\mathit{erms}(\Sigma, \mathcal{V})$. $\mathit{Var}(s)$ is the set of variables in s. A term s is *ground* if $\mathit{Var}(s) = \emptyset$.

Contexts and Substitution. Fixing an environment Γ, a *substitution* is a mapping $[x_1 := s_1, \ldots, x_k := s_k]$ from variables to terms, with $\{x_1 : \iota_1, \ldots, x_k : \iota_k\} \subseteq \Gamma$ and where $\Gamma \vdash s_1 : \iota_1, \ldots, \Gamma \vdash s_k : \iota_k$. The result $s\gamma$ of applying a substitution γ on a term s, is s with all occurrences of any x_i replaced by s_i. A *context* C is a term with zero or more special variables: $\Box_1, \ldots, \Box_n$, each occurring once. If $\Gamma \cup \{\Box_1 : \iota_1, \ldots, \Box_n : \iota_n\} \vdash C : \kappa$, and also $\Gamma \vdash s_i : \iota_i$ for all i, then we define the term $C[s_1, \ldots, s_n]$ as C with each $\Box_i$ replaced by the corresponding s_i.

Rules and Rewriting. In a many-sorted TRS (without constraints!) rules are pairs $l \to r$ where l and r are terms, l is not a variable and $\mathit{Var}(r) \subseteq \mathit{Var}(l)$; moreover, l and r must have the same sort. A (finite or infinite) set of rules $\mathcal{R}$ induces a rewrite relation $\to_{\mathcal{R}}$ on the set of terms by the following inference rule:

$$C[l\gamma] \to_{\mathcal{R}} C[r\gamma] \text{ for all rules } l \to r \text{, contexts } C \text{ and substitutions } \gamma.$$

$\to_{\mathcal{R}}^{+}$ denotes the transitive closure of $\to_{\mathcal{R}}$, and $\to_{\mathcal{R}}^{*}$ the reflexive-transitive one.

Example 2. We consider a many-sorted TRS, with signature and rules as follows:

$$\begin{array}{lll}
\mathsf{0} : \mathsf{int} & \mathsf{plus} : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int} & \mathsf{geq2} : [\mathsf{int} \times \mathsf{int} \times \mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{bool} \\
\mathsf{s} : [\mathsf{int}] \Rightarrow \mathsf{int} & \mathsf{sum} : [\mathsf{int}] \Rightarrow \mathsf{int} & \mathsf{sum2} : [\mathsf{bool} \times \mathsf{int}] \Rightarrow \mathsf{int} \\
\mathsf{p} : [\mathsf{int}] \Rightarrow \mathsf{int} & \mathsf{geq} : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{bool} &
\end{array}$$

$$\begin{array}{rlrl}
\mathsf{sum}(x) & \to \mathsf{sum2}(\mathsf{geq}(\mathsf{0}, x), x) & \mathsf{geq}(x, y) & \to \mathsf{geq2}(x, y, \mathsf{0}, \mathsf{0}) \\
\mathsf{sum2}(\mathsf{true}, x) & \to \mathsf{0} & \mathsf{geq2}(\mathsf{s}(x), y, z, u) & \to \mathsf{geq2}(x, y, \mathsf{s}(z), u) \\
\mathsf{sum2}(\mathsf{false}, \mathsf{s}(x)) & \to \mathsf{plus}(\mathsf{s}(x), \mathsf{sum}(x)) & \mathsf{geq2}(\mathsf{p}(x), y, z, u) & \to \mathsf{geq2}(x, y, z, \mathsf{s}(u)) \\
\mathsf{plus}(\mathsf{s}(x), y) & \to \mathsf{s}(\mathsf{plus}(x, y)) & \mathsf{geq2}(\mathsf{0}, \mathsf{s}(x), y, z) & \to \mathsf{geq2}(\mathsf{0}, x, y, \mathsf{s}(z)) \\
\mathsf{plus}(\mathsf{p}(x), y) & \to \mathsf{p}(\mathsf{plus}(x, y)) & \mathsf{geq2}(\mathsf{0}, \mathsf{p}(x), y, z) & \to \mathsf{geq2}(\mathsf{0}, x, \mathsf{s}(y), z) \\
\mathsf{plus}(\mathsf{0}, y) & \to y & \mathsf{geq2}(\mathsf{0}, \mathsf{0}, \mathsf{s}(x), \mathsf{s}(y)) & \to \mathsf{geq2}(\mathsf{0}, \mathsf{0}, x, y) \\
\mathsf{s}(\mathsf{p}(x)) & \to x & \mathsf{geq2}(\mathsf{0}, \mathsf{0}, x, \mathsf{0}) & \to \mathsf{true} \\
\mathsf{p}(\mathsf{s}(x)) & \to x & \mathsf{geq2}(\mathsf{0}, \mathsf{0}, \mathsf{0}, \mathsf{s}(x)) & \to \mathsf{false}
\end{array}$$

Here, $\mathsf{sum}(n)$ calculates $\Sigma_{i=1}^{n} i$. Because we consider integers, rather than the (well-founded) natural numbers, this is a somewhat tricky system for common analysis methods. A term $\mathsf{sum}(\mathsf{s}(\mathsf{0}))$ is reduced to $\mathsf{s}(\mathsf{0})$ in 11 steps.

Example 3. We might simplify Example 2 by considering an infinite signature, which contains all integers, and an infinite set of rules, as is (roughly) done in [8]:

$$\begin{array}{rll}
\mathsf{sum}(x) & \to \mathsf{sum2}(\mathsf{geq}(\mathsf{0}, x), x) & \\
\mathsf{sum2}(\mathsf{true}, x) & \to \mathsf{0} & \\
\mathsf{sum2}(\mathsf{false}, x) & \to \mathsf{plus}(x, \mathsf{sum}(\mathsf{plus}(x, -\mathsf{1}))) & \\
\mathsf{plus}(\mathsf{n}, \mathsf{m}) & \to \mathsf{k} & \forall n, m, k \in \mathbb{Z} \text{ such that } n + m = k \\
\mathsf{geq}(\mathsf{n}, \mathsf{m}) & \to \mathsf{true} & \forall n, m \in \mathbb{Z} \text{ such that } n \geq m \\
\mathsf{geq}(\mathsf{n}, \mathsf{m}) & \to \mathsf{false} & \forall n, m \in \mathbb{Z} \text{ such that } n < m
\end{array}$$

This system is more pleasant, but has infinitely many rules, which makes it awkward to deal with except for dedicated techniques. Also, we still have to encode the constraints in the rules (and add rules to evaluate them), which makes analysis tricky. For example, termination of $\mathsf{sum}(x)$ relies on x getting closer to 0 in every step; to prove this, we must track the implications of $\mathsf{geq}(\mathsf{0}, x) \to_{\mathcal{R}}^{*} \mathsf{false}$.

Note: term rewriting is usually defined without sorts. Then, function symbols have an *arity* (number of arguments) rather than a sort declaration. Such a TRS can be seen as a many-sorted TRS by assigning to symbols with arity n a sort declaration $[\mathsf{term} \times \ldots \times \mathsf{term}] \Rightarrow \mathsf{term}$, with n occurrences of term before the $\Rightarrow$.

3 Term Rewriting with Logical Constraints

Examples 2 and 3 illustrate why rewriting with native support for integer operations and constraints is a good idea. Normal rewriting simply does not seem adequate when handling data types which are not usually defined inductively.

We could add integers and integer constraints to rewriting, as in [3]. But with equal effort, we may be more general. Rather than focusing on $\mathbb{Z}$, we follow the ideas of [9] and take the underlying domain, and operations on it, as parameters.

Terms. We assume given a signature $\Sigma = \Sigma_{terms} \cup \Sigma_{theory}$. *Terms* are elements of $\mathcal{T}erms(\Sigma, \mathcal{V})$ as in Section 2.2. Moreover, we assume given a mapping $\mathcal{I}$ which assigns to each sort occurring in Σ_{theory} a set, and an *interpretation mapping* $\mathcal{J}$ which maps each $f : [\iota_1 \times \ldots \times \iota_n] \Rightarrow \kappa \in \Sigma_{theory}$ to a function $\mathcal{J}_f$ in $\mathcal{I}_{\iota_1} \Longrightarrow \ldots \Longrightarrow \mathcal{I}_{\iota_n} \Longrightarrow \mathcal{I}_{\kappa}$. For every sort ι occurring in Σ_{theory} we also fix a set $\mathcal{V}al_{\iota} \subseteq \Sigma_{theory}$ of *values*: function symbols $a : [] \Rightarrow \iota$, where $\mathcal{J}$ gives a one-to-one mapping from $\mathcal{V}al_{\iota}$ to $\mathcal{I}_{\iota}$. Let $\mathcal{V}al$ be the set of all values. We generally identify a value c with the logical term $c()$. An interpretation mapping can be extended to an interpretation on ground terms in $\mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$ in the obvious way:

$$[\![f(s_1, \ldots, s_n)]\!]_{\mathcal{J}} = \mathcal{J}_f([\![s_1]\!]_{\mathcal{J}}, \ldots, [\![s_n]\!]_{\mathcal{J}})$$

The elements of Σ_{theory} and Σ_{terms} may overlap only on values ($\Sigma_{theory} \cap \Sigma_{terms} \subseteq \mathcal{V}al$). We call a term in $\mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$ a *logical term*, and a term in $\mathcal{T}erms(\Sigma_{terms}, \mathcal{V})$ a *proper term*. Intuitively, logical terms define a function or constraint in the model, while proper terms are the objects we want to rewrite. Mixed terms typically occur as intermediate steps in a reduction.

A ground logical term s *has value* t if t is a value such that $[\![s]\!] = [\![t]\!]$. Every ground logical term has a unique value. A *logical constraint* is a logical term of some sort bool with $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$. We generally let $\mathcal{V}al_{\mathsf{bool}} = \{\mathsf{true}, \mathsf{false}\}$. A ground logical constraint s is *valid* if $[\![s]\!]_{\mathcal{J}} = \top$. A non-ground constraint s is valid if $s\gamma$ is valid for all substitutions γ which map the variables in $Var(s)$ to a value.

Example 4. Choosing $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$ and $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$, we might let Σ_{theory} be the set below, with interpretations as given in e.g. SMTLIB [1]:

$$\begin{array}{rll}
\mathsf{true} : \mathsf{bool} & + : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int} & \wedge : [\mathsf{bool} \times \mathsf{bool}] \Rightarrow \mathsf{bool} \\
\mathsf{false} : \mathsf{bool} & \geq : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{bool} & \neg : [\mathsf{bool}] \Rightarrow \mathsf{bool} \\
\mathsf{n} : \mathsf{int}\ (n \in \mathbb{Z}) & = : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{bool} &
\end{array}$$

Moreover, we let $\Sigma_{terms} = \{\mathsf{sum} : [\mathsf{int}] \Rightarrow \mathsf{int}\} \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\}$.

The values in Σ are true, false, and n for all $n \in \mathbb{Z}$. Examples of logical terms, considering $\geq$, $+$ and $=$ as infix symbols, are $\mathsf{0} = \mathsf{0} + \mathsf{-1}$ and $x + \mathsf{3} \geq y + \mathsf{-42}$. Both are constraints. $\mathsf{5} + \mathsf{9}$ is also a ground logical term, but is not a constraint. $\mathsf{sum}(x)$ and $\mathsf{sum}(\mathsf{sum}(\mathsf{42})$ are proper terms. The value $\mathsf{0}$ is both a proper and a logical term. $\mathsf{sum}(\mathsf{37} + \mathsf{5})$ is neither, but is still a term (also called *mixed term*).

In Example 4 we restricted interest to functions in SMTLIB for Σ_{theory}, but this is not fundamental; we might also for instance have a symbol $\mathsf{p} : [\mathsf{int}] \Rightarrow \mathsf{int}$ with $\mathcal{J}_{\mathsf{p}} = \lambda x.x - 1$, or $\mathsf{pi} : [\mathsf{int}] \Rightarrow \mathsf{int}$ with $\mathcal{J}_{\mathsf{pi}} = \lambda n.$"the n^{th} decimal of π". It is in general a good idea, however, to limit interest to computable functions.

Rules and Rewriting. A *rule* is a triple $l \to r\ [\varphi]$ where l and r are terms, and φ is a logical constraint. l must have the form $f(l_1, \dots, l_n)$ with $f \in \Sigma_{terms} \backslash \Sigma_{theory}$, and l and r must have the same sort. If $\varphi = \mathsf{true}$ with $\mathcal{J}(\mathsf{true}) = \top$, the rule is usually just denoted $l \to r$. A rule is *regular* if $Var(\varphi) \subseteq Var(l)$ and *standard* if l is a proper term. We define $LVar(l \to r\ [\varphi]))$ as $Var(\varphi) \cup (Var(r) \setminus Var(l))$.

A substitution γ *respects* a rule $l \to r\ [\varphi]$ if $Dom(\gamma) = Var(l) \cup Var(r) \cup Var(\varphi)$, $\gamma(x)$ is a value for all $x \in LVar(l \to r\ [\varphi])$ and $\varphi\gamma$ is valid.

We assume given a set of rules $\mathcal{R}$. The *rewrite relation* $\to_{\mathcal{R}}$ is a relation on terms, defined as the union of $\to_{\mathtt{rule}}$ and $\to_{\mathtt{calc}}$, where:

$$\begin{array}{rcl}
C[l\gamma] \to_{\mathtt{rule}} C[r\gamma] & & \text{if } l \to r\ [\varphi] \in \mathcal{R} \text{ and } \gamma \text{ respects } l \to r\ [\varphi] \\
C[f(s_1, \dots, s_n)] \to_{\mathtt{calc}} C[v] & & \text{if } f \in \Sigma_{theory} \backslash \Sigma_{terms},\ \text{all } s_i \text{ values,} \\
 & & \quad \text{and } v \text{ is the value of } f(s_1, \dots, s_n)
\end{array}$$

A reduction step with $\to_{\mathtt{calc}}$ is called a *calculation*. A term is in *normal form* if (and only if) it cannot be reduced with $\to_{\mathcal{R}}$. Sometimes we consider *innermost reduction*: $C[f(\boldsymbol{s})] \to_{\mathcal{R},in} C[t]$ if $f(\boldsymbol{s}) \to_{\mathcal{R}} t$, and all s_i are in normal form.

A *logical constrained term rewriting system* (LCTRS) is defined as the pair $(\mathcal{T}erms(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$. An LCTRS is typically given by providing $\mathcal{I}$ and $\mathcal{J}$ and the sets $\Sigma_{terms}, \Sigma_{theory}$ and $\mathcal{R}$. When clear from context, the signatures and mappings may be omitted. An *innermost LCTRS* is the pair $(\mathcal{T}erms(\Sigma, \mathcal{V}), \to_{\mathcal{R},in})$.

A (normal or innermost) LCTRS is *standard* or *regular* if all its rules are standard or regular respectively. In a regular LCTRS, $\to_{\mathcal{R}}$ is computable, provided $\mathcal{J}_f$ is computable for all $f \in \Sigma_{theory}$. Even in a regular LCTRS, the right-hand sides of rules may contain fresh variables. This can for example be used to simulate user input. Think for example of a rule $\mathsf{Start} \to \mathsf{Handle}(input)$ where $input$ is a variable; by definition of $\to_{\mathcal{R}}$, $input$ can only be instantiated by a value.

Example 5. It is time to see how these definitions work in practice. Let us modify Example 2 to use constraints and calculations. We have defined Σ_{theory} and Σ_{terms} in Example 4. The rules are replaced by the following set:

$$\mathsf{sum}(x) \to \mathsf{0}\ [\mathsf{0} \geq x] \qquad \mathsf{sum}(x) \to x + \mathsf{sum}(x + -\mathsf{1}) \quad [\neg(\mathsf{0} \geq x)]$$

Note that the sum rules may only be applied to a term $\mathsf{sum}(n)$ whose immediate argument n is a value, so this subterm itself cannot contain the symbol sum.

For an example derivation, let us calculate $\Sigma_{n=1}^{2} n$. We have: $\mathsf{sum}(\mathsf{2}) \to_{\mathtt{rule}} \mathsf{2} + \mathsf{sum}(\mathsf{2} + -\mathsf{1}) \to_{\mathtt{calc}} \mathsf{2} + \mathsf{sum}(\mathsf{1}) \to_{\mathtt{rule}} \mathsf{2} + (\mathsf{1} + \mathsf{sum}(\mathsf{1} + -\mathsf{1})) \to_{\mathtt{calc}} \mathsf{2} + (\mathsf{1} + \mathsf{sum}(\mathsf{0})) \to_{\mathtt{rule}} \mathsf{2}+(\mathsf{1}+\mathsf{0}) \to_{\mathtt{calc}} \mathsf{2}+\mathsf{1} \to_{\mathtt{calc}} \mathsf{3}$. In each step, it so happens that we have exactly one choice of what rule to apply, and where. For example in the first step, $\neg(\mathsf{0} \geq \mathsf{2})$ holds and $\mathsf{0} \geq \mathsf{2}$ does not, so only the second rule is applicable. Neither rule can be applied on $\mathsf{sum}(\mathsf{2} + -\mathsf{1})$, as $\mathsf{2} + -\mathsf{1}$ is no value; $\to_{\mathtt{calc}}$ *is* applicable. We cannot use a calculation step on $\mathsf{2} + (\mathsf{1} + \mathsf{sum}(\mathsf{0}))$, as there is no subterm with the right form; the system does not know about associativity.

One might wonder why we insist that all variables in $LVar$ are instantiated with values, rather than just logical terms. Having a rule $f(x, y) \to y\ [x \geq y]$, could we not reduce $f(x + \mathsf{1}, x)$ without this instantiation?

The reason to require that $\gamma(x)$ is ground for $x \in Var(\varphi)$ is simplicity of the rewrite relation: by posing this restriction, validity of $\varphi\gamma$ is mostly easy to test. Without it, validity might not be computable.[1] Moreover, without this restriction the reduction relation is not preserved under substitution: for a symbol $\mathsf{a} \in \Sigma_{terms} \setminus \Sigma_{theory}$, we cannot have $f(\mathsf{a} + \mathsf{1}, \mathsf{a}) \to \mathsf{a}$, as $\mathsf{a} + \mathsf{1}$ is not a logical term. It does make sense to study whether a term $f(x + \mathsf{1}, y)$, or even a term $f(x, y)$ with $x > y$ reduces. In Section 4 we will see how to rewrite *constrained terms*.

By requiring that $\gamma(x)$ is even a value, we avoid complicating notions like complexity. If we could $\to_{\mathtt{rule}}$-reduce terms like $\mathsf{sum}(\mathsf{3}+-\mathsf{1}+-\mathsf{1}+-\mathsf{1})$, then these additions would have to be calculated in every step when testing the constraint. There does not seem to be any advantage to allowing these hidden calculations; we can simply $\to_{\mathtt{calc}}$-normalise ground logical terms before applying a rule step. For the same motivation of complexity, we only allow $\to_{\mathtt{calc}}$ to take single steps, rather than allowing $C[s] \to_{\mathtt{calc}} C[v]$ for any logical term s with value v.

Note that $\to_{\mathtt{calc}}$ is not special; using $\to_{\mathtt{calc}}$ is functionally equivalent to extending $\mathcal{R}$ with all rules $f(x_1, \dots, x_n) \to y\ [f(x_1, \dots, x_n) = y]$ where $f \in \Sigma_{theory}$.

Regularity and Standardness. Regularity is a useful condition. An irregular LCTRS is not in general deterministic, polynomially solvable, or even computable. Consider for example a rule $\mathsf{f}(x) \to \mathsf{g}(\mathsf{f}(y), \mathsf{f}(z))\ [x = y{*}z{\wedge}y > \mathsf{1}{\wedge}z > \mathsf{1}]$, which quickly decomposes a natural number into its prime factors.

Still, there is some advantage in allowing irregular systems. For example, in termination analysis a transformation might chain the two regular rules $f(x) \to g(x - \mathsf{1}, input)\ [x > \mathsf{0}]$ and $g(x, y) \to f(y)\ [x \geq y]$ into a single irregular rule: $f(x) \to f(y)\ [x > \mathsf{0} \wedge x - \mathsf{1} \geq y]$. In addition, an irregular rule can be used to

[1] For example, defining $\mathcal{J}_p(n)$ to be true if a sequence of 9999 nines starts at the n^{th} decimal of π, and false otherwise, and considering a rule $f(x) \to x\ [\neg p(x)]$, we don't know whether a term $f(x)$ should reduce for all instances of x.

calculate a partial function, e.g. $\mathsf{div}(x, y) \to z\ [z * y = x]$, which cannot be easily defined otherwise. Note that such a rule does not lead to undecidability.

Similar to regularity, *standardness* is convenient: in a standard system there are no overlaps between $\to_{\mathtt{rule}}$ and $\to_{\mathtt{calc}}$. Standardness is a natural property, since symbols from $\Sigma_{theory} \setminus \Sigma_{terms}$ in terms are conceptually intended primarily as a way to do calculations. We don't use modulo reasoning: a non-standard rule $f(x + \mathsf{1}) \to r$ matches only terms of the form $f(s + \mathsf{1})$, so *not* for instance $f(\mathsf{3})$. As with regularity, non-standard systems may arise during analysis, for example when a rule is reversed. Note that even in standard systems, left-hand sides of rules may contain values (or other symbols in Σ_{terms}); while we often encounter rules of the form $f(x_1, \ldots, x_n) \to r\ [\varphi]$, this is not an innate property.

Overview. Compared to the rather hairy (Example 2) or infinite (Example 3) systems obtained when encoding integer arithmetic and constraints in a normal TRS, LCTRSs offer an elegant alternative. Although LCTRSs often have infinite signatures, calculation steps make it possible to avoid infinite sets of rules.

Example 6. To demonstrate a situation where we should not use the integers as an underlying set, consider the following short imperative program:

```
function main() {
  byte x = input();
  while (x < 150) {
    if (x == 0) x = 1;
    x = x * 2;
  }
  return x;
}
```

Here a `byte` is an unsigned 8-bit integer. This program doesn't terminate: input 0 gives an infinite reduction (with `x` changing from 0 to 1, 2, 4, 8, 16, 32, 64, 128, 0).

Using the ideas of [3], we might model this program as follows:

$$\begin{array}{rll@{\quad}ll}
\mathsf{main} \to \mathsf{loop}(\mathit{input}) & & & \mathsf{loop}_1(x) \to \mathsf{loop}_2(\mathsf{1}) & [x = \mathsf{0}] \\
\mathsf{loop}(x) \to \mathsf{loop}_1(x) & [x < 150] & & \mathsf{loop}_1(x) \to \mathsf{loop}_2(x) & [\neg(x = \mathsf{0})] \\
\mathsf{loop}(x) \to \mathsf{return}(x) & [\neg(x < 150)] & & \mathsf{loop}_2(x) \to \mathsf{loop}(x * \mathsf{2}) &
\end{array}$$

As bytes are not unbounded, our underlying set is *not* $\mathbb{Z}$. Rather, we consider the set $\mathcal{BV}_8$ of bit vectors of length 8, and let $\mathcal{J}$ map to corresponding notions of addition, multiplication, comparison and equality (see SMT-LIB [1]). With this theory, a reduction from start adequately simulates a reduction in the original imperative program. Note that if we had naively translated the program to an LCTRS over the integers, we could have falsely concluded (local) termination.

We can analyse systems on bitvectors in much the same way as we analyse systems over the integers. We will see some ideas for this in Section 6.

4 Constrained Terms

As discussed in Section 3, there are good reasons why the definition of rewriting requires that variables in a constraint are instantiated by values. But sometimes you may want to know whether a term of a certain form rewrites. For example, if we know that $x < -\mathsf{3}$, this is enough to decide that $\mathsf{sum}(x)$ reduces to $\mathsf{0}$.

In this section, we will therefore consider *constrained terms*: pairs $s\ [\varphi]$ of a term s and a constraint φ. Constrained terms are harder to rewrite and analyse than normal terms, but sometimes the need may arise. For instance in rewriting induction (see e.g. [4,9]) when proving that $f(x) \leftrightarrow^* g(x, \mathsf{0})$ $[x \geq \mathsf{1}]$, being able to reason about the reducts of $f(x)$ $[x \geq \mathsf{1}]$ is very relevant.

To rewrite constrained terms, we must take several things into account. For example, given a rule $f(\mathsf{0}) \to \mathsf{1}$, we should be able to reduce $f(x)$ $[x = \mathsf{0}]$, even though $f(x)$ itself is not matched by the left-hand side. We will also need to deal with irregular rules; given a rule $f(x) \to g(y)$ $[y > x]$, we should be able to reduce $f(x)$ $[x > \mathsf{3}]$ to $g(y)$ $[y > \mathsf{4}]$, or at least to an instance, like $g(y)$ $[x > \mathsf{3} \wedge y = x + \mathsf{1}]$.

A substitution γ *respects* a constrained term s $[\varphi]$ if $\gamma(x)$ is a value for all $x \in \mathit{Var}(\varphi)$ and $[\![\varphi\gamma]\!] = \top$. Two constrained terms s $[\varphi]$ and t $[\psi]$ are *equivalent*, notation $s\ [\varphi] \approx t\ [\psi]$, if for all substitutions γ which respect s $[\varphi]$ there is a substitution δ which respects t $[\psi]$ such that $s\gamma = t\delta$, and vice versa.

Example 7. Examples of constrained terms over the signature of sum are:

1. $\mathsf{sum}(x)$ $[x \geq \mathsf{3}]$;
2. $x + y$ $[x \geq y \wedge \neg(x = y) \wedge x = \mathsf{3}]$;
3. $\mathsf{3} + z$ $[1 \geq x \wedge x + \mathsf{1} \geq z]$;

Constrained terms 2 and 3 are equivalent, as the following formulas hold in $\mathbb{Z}$:

$$\forall x, y.(x \geq y \wedge \neg(x = y) \wedge x = 3) \Rightarrow \exists x', z.1 \geq x' \wedge x' + 1 \geq z \wedge x = 3 \wedge y = z$$
$$\forall x', z.(1 \geq x' \wedge x' + 1 \geq z) \Rightarrow \exists x, y.x \geq y \wedge \neg(x = y) \wedge x = 3 \wedge y = z$$

It is clear that equivalence of two constrained terms is not always easy to tell.

To be able to modify constraints, we assume that Σ_{theory} contains a symbol $\wedge : [\mathsf{bool} \times \mathsf{bool}] \Rightarrow \mathsf{bool}$ with $\mathcal{J}_\wedge$ the conjunction operator on the booleans, and for all sorts ι a symbol $=_\iota$: $[\iota \times \iota] \Rightarrow \mathsf{bool}$ with $\mathcal{J}_{=_\iota} := \lambda n, m \in \mathcal{I}_\iota.n = m$.

Rewriting Constrained Terms. We let $s\ [\varphi] \to_{\mathtt{calc}} t\ [\varphi \wedge x = f(s_1, \dots, s_n)]$ if $s = C[f(s_1, \dots, s_n)]$ with $f \in \Sigma_{theory} \setminus \Sigma_{terms}$, all s_i in $\mathit{Var}(\varphi) \cup \mathcal{V}al$, x a fresh variable, and $t = C[x]$. Additionally, $s\ [\varphi] \to_{\mathtt{rule}} t\ [\varphi]$ if φ is satisfiable, $s = C[l\gamma]$ and $t = C[r\gamma]$ for some rule $l \to r\ [\psi]$ and substitution γ such that:

- $\mathit{Dom}(\gamma) = \mathit{Var}(l) \cup \mathit{Var}(r) \cup \mathit{Var}(\psi)$
- $\gamma(x)$ is a value or variable in $\mathit{Var}(\varphi)$ for all $x \in \mathit{LVar}(l \to r\ [\psi])$
- $\varphi \Rightarrow (\psi\gamma)$ is valid (that is, for all δ with $\varphi\delta$ valid also $\psi\gamma\delta$ is valid)

The relation $\to_{\mathcal{R}}$ on constrained terms is defined as $\approx \cdot (\to_{\mathtt{calc}} \cup \to_{\mathtt{rule}}) \cdot \approx$.

Example 8. With the rule $f(\mathsf{0}) \to \mathsf{1}$: $f(x)\ [x = \mathsf{0}] \approx f(\mathsf{0})\ [\mathsf{true}] \to_{\mathtt{rule}} \mathsf{1}\ [\mathsf{true}]$.

Example 9. With the irregular rule $f(x) \to g(y)\ [y > x]$, we have: $f(x)\ [x > \mathsf{3}] \approx f(x)\ [x > \mathsf{3} \wedge y > x] \to_{\mathtt{rule}} g(y)\ [x > \mathsf{3} \wedge y > x] \approx g(y)\ [y > \mathsf{4}]$. Similarly, $f(x)\ [x > \mathsf{0}]$ reduces with $f(x) \to g(y)\ [x = y + \mathsf{1}]$ to $f(y)\ [y \geq \mathsf{0}]$. We do *not* have that $f(x)\ [\mathsf{true}] \to g(x - \mathsf{1})\ [\mathsf{true}]$, as $x - \mathsf{1}$ cannot be instantiated to a value.

Example 10. Following on Example 5, we may reduce $\mathsf{sum}(x)\ [x > \mathsf{2}]$ as follows:

$$\begin{aligned}
\mathsf{sum}(x)\ [x > \mathsf{2}] &\to_{\mathcal{R}} x + \mathsf{sum}(x + -\mathsf{1})\ [x > \mathsf{2}] \\
&\to_{\mathcal{R}} x + \mathsf{sum}(y)\ [x > \mathsf{2} \wedge y = x + -\mathsf{1}] \\
&\to_{\mathcal{R}} x + (y + \mathsf{sum}(y + -\mathsf{1}))\ [x > \mathsf{2} \wedge y = x + -\mathsf{1}] \\
&\to_{\mathcal{R}} x + (y + \mathsf{sum}(z))\ [x > \mathsf{2} \wedge y = x + -\mathsf{1} \wedge z = y + -\mathsf{1}]
\end{aligned}$$

The notion of reduction on constrained terms is intimately tied to the notion of reduction on terms, as the following two theorems demonstrate:

Theorem 1. *If $s \to_{\mathcal{R}} t$ then also $s\ [\mathsf{true}] \to_{\mathcal{R}} t\ [\mathsf{true}]$.*

Theorem 2. *If $s\ [\varphi] \to_{\mathcal{R}} t\ [\psi]$ then for all substitutions γ which respect $s\ [\varphi]$ there is a substitution δ which respects $t\ [\psi]$ such that $s\gamma \to_{\mathcal{R}} t\delta$.*

Thus, we have a notion of constrained terms and reduction thereof. We do not consider these notions as basic; rather, using for instance Theorem 2, they can be used in analysis to find properties of unconstrained terms in the system (think for instance of an inductive proof that $\mathsf{sum}(x) \geq x$ if $x \geq \mathsf{0}$).

Determining whether a constrained term reduces, or what it reduces to, is a difficult problem. In special cases (for instance, regular rules with linear integer arithmetic) it is decidable, but in others we may have to resort to clever guessing.

5 Comparison to Existing Systems

So, we have a new formalism. Is this truly more convenient or general than existing formalisms? Here we will briefly study some formalisms from the literature, and sketch how they relate to the LCTRSs introduced here. However, a comprehensive study of all relevant formalisms is beyond the reach of this paper.

5.1 Constrained TRSs from [9,12,14]

LCTRSs are based primarily on the constrained TRSs in [9,12,14]. Like our LCTRSs, these systems have a separate theory signature, and a given interpretation mapping $\mathcal{J}$. The main difference is that they have no values or calculation steps. Instead, these features are encoded in the terms and rules, with for example the integers being represented as $\mathsf{0}, \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\mathsf{0})), \dots, \mathsf{p}(\mathsf{0}), \mathsf{p}(\mathsf{p}(\mathsf{0})), \dots$

Example 11. The sum system is implemented as a CTRS with rules:

$$\begin{array}{rlrl} \mathsf{sum}(x) \to \mathsf{0} & [\mathsf{0} \geq x] & \mathsf{0} + y \to y & \mathsf{s}(\mathsf{p}(x)) \to x \\ \mathsf{sum}(\mathsf{s}(x)) \to \mathsf{sum}(x) + \mathsf{s}(x) & [x \geq \mathsf{0}] & \mathsf{s}(x) + y \to \mathsf{s}(x + y) & \mathsf{p}(\mathsf{s}(x)) \to x \\ & & \mathsf{p}(x) + y \to \mathsf{p}(x + y) & \end{array}$$

Compared to their predecessors, LCTRS are far simpler to use: by having symbols for all values and using calculation steps, systems are implemented much more concisely (as demonstrated by sum). Moreover, the resulting systems are easier to analyse. For example, note that this version of sum is not a constructor system, and proving confluence or complete reducibility is difficult. Also, due to the countable nature of terms, no finite CTRS can encode Example 12:

Example 12. Using sorts int, real and bool, and addition on the real numbers denoted by $+.$, we might represent the function $n \mapsto \Sigma_{i=i}^{n} \sqrt{n}$ as follows:

$$\mathsf{sumroot}(x) \to \mathsf{0.0}\ [\mathsf{0} \geq x] \qquad \mathsf{sumroot}(x) \to \mathsf{sqrt}(x) +.\ \mathsf{sumroot}(x - \mathsf{1})\ [\neg(\mathsf{0} \geq x)]$$

There does not seem to be an easy way to simulate CTRSs as LCTRSs or the other way around. However, initial results suggest that results for CTRSs [9,12,14] easily extend to LCTRSs, and are moreover vastly simplified by the translation.

5.2 Integer Term Rewriting Systems

In Example 3 we saw a system somewhat like the *integer term rewriting systems* in [8]. These ITRSs are innermost TRSs with an infinite signature $\Sigma \cup \Sigma_{int}$, where Σ_{int} includes $BOp = \{+, -, *, /, \%, >, \geq, <, \leq, =, \neq, \wedge, \Rightarrow\}$ and moreover true, false and all integers. $\mathcal{R}$ is defined as $R \cup \mathcal{PD}$, where $\mathcal{PD} = \{n \circ m \to k \mid n, m, k \in \mathbb{Z} \cup \mathbb{B},\ \circ \in BOp \mid n \circ m = k \text{ holds in } \mathbb{Z} \text{ and } \mathbb{B}\}$ (e.g. $1 + 2 \to 3 \in \mathcal{PD}$).

Example 13. An example ITRS in [8] has $\Sigma = \{\mathsf{log}, \mathsf{lif}\}$ and R consisting of:

$$\begin{array}{ll} \mathsf{log}(x, y) \to \mathsf{lif}(x \geq y \wedge y > 1, x, y) & \mathsf{lif}(\mathsf{true}, x, y) \to 1 + \mathsf{log}(x/y, y) \\ & \mathsf{lif}(\mathsf{false}, x, y) \to 0 \end{array}$$

Terms containing symbols $\geq$, $\wedge$, $+$ and $/$ can be rewritten using the $\mathcal{PD}$ rules.

We can model an ITRS as an LCTRS, which is finite if R is finite. ITRSs are not defined with sorts, but sorts can easily be imagined. Indeed, there seems little reason to analyse the behaviour of a term like $\mathsf{log}(\mathsf{true}, 5 + \mathsf{false})$, and for innermost termination (the primary area of interest for ITRSs) presence of sorts makes no difference [7]. The only other issue is that some elements of BOp ($/$ and $\%$) define *partial* functions, so cannot be modelled by calculations.

To define an LCTRS with the same terms and rewrite relation as a given *sorted* ITRS (with sorts int and bool assigned in the obvious way), let $\mathcal{V}al := \{\mathsf{true}, \mathsf{false} : \mathsf{bool}\} \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\}$, $\Sigma_{terms} := \Sigma \cup \mathcal{V}al \cup \{/, \% : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int}\}$ and $\Sigma_{theory} := \mathcal{V}al \cup (BOp \setminus \{/, \%\})$. We use the expected interpretations for Σ_{theory}. Let $\mathcal{R} := R \cup \{x/y \to z\ [x = y*z],\ x\%y \to z\ [x = y*u+z \wedge 0 \leq z \wedge z < y]\}$. Then $\to_{\mathcal{R}}$ is exactly the reduction relation from the original ITRS. $\mathcal{R}$ is finite, because $\to_{\mathtt{calc}}$ and the two irregular rules take over the role of $\to_{\mathcal{PD}}$. Note that the irregularity is not an issue for computability in this case.

Example 14. The system from Example 13 becomes the following LCTRS (ignoring the $\%$ symbol which does not occur in any rule):

$$\begin{array}{ll} \mathsf{log}(x, y) \to \mathsf{lif}(x \geq y \wedge y > 1, x, y) & \mathsf{lif}(\mathsf{true}, x, y) \to 1 + \mathsf{log}(x/y, y) \\ x/y \to z\ [x = y * z] & \mathsf{lif}(\mathsf{false}, x, y) \to 0 \end{array}$$

Comment: if, for whatever reason, we do want to analyse the original *unsorted* ITRS, we can also do so with an LCTRS. In this case, we assign a single sort term as suggested in Section 2, and let $\mathcal{I}_{\mathsf{term}} = \mathbb{Z} \cup \mathbb{B}$; we cannot use calculations now, because all functions are partial, but can encode $\to_{\mathcal{PD}}$ with irregular rules.

Conditional ITRSs. Integer TRSs play a role in the termination analysis of Java Bytecode employed in [13]. There, termination of JBC is reduced to termination of a *conditional* ITRS; rules look somewhat like the rules in LCTRSs:

$$\begin{array}{ll} \mathsf{log}(x, y) \to 1 + \log(x/y, y) & \mid\ x \geq y \wedge y > 1 \to^* \mathsf{true} \\ \mathsf{log}(x, y) \to 0 & \mid\ \neg(x \geq y \wedge y > 1) \to^* \mathsf{true} \end{array}$$

These systems are unravelled to ITRSs for analysis (giving the system from Example 13). However, if the elements of Σ_{int} are not root symbols of left-hand sides of R, and $/$ and $\%$ do not occur in the conditions, we can translate such systems into LCTRSs immediately (replacing conditions by constraints), and obtain a system where constraints are not encoded. We can prove that a CITRS generates the same relation as its transformation to an innermost LCTRS.

As for the other direction, LCTRSs are not a special case of ITRS. Most importantly, ITRSs have no native treatment of constraints. These have to be encoded, and to for instance prove termination of even simple systems we need far more powerful techniques than in the LCTRS setting. Moreover, ITRSs are restricted to the integers. While Example 6 *can* be encoded, using rules like $\mathsf{loop}_2(x) \to \mathsf{loop}((x * 2)\%256)$, ITRSs cannot represent for instance Example 12.

5.3 $\mathbb{Z}$-TRSs

Next, we consider a mixture of the ideas in [3,4,5,6].[2] $\mathbb{Z}$-TRSs are based on a many-sorted signature Σ, which must include $\Sigma_{\mathtt{int}} = \{0, 1 : \mathsf{int},\ - : [\mathsf{int}] \Rightarrow \mathsf{int},\ +, * : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int}\}$. *Constraints* are given by the grammar:

$$\mathsf{C} ::= \mathsf{true} \mid \mathsf{false} \mid s = t \mid s > t \mid s \geq t \mid \neg \mathsf{C} \mid \mathsf{C} \wedge \mathsf{C} \qquad s, t \text{ terms over } \Sigma_{\mathtt{int}}$$

Rules are triples $l \to r\ [\varphi]$ where φ is a constraint, l and r are terms with the same sort, and l has the form $f(l_1, \dots, l_n)$. The left-hand sides may not contain any element of $\Sigma_{\mathtt{int}}$. The reduction relation is defined much like in LCTRSs, except that γ "respects" $l \to r\ [\varphi]$ if all variables of sort int are instantiated by (not necessarily ground) terms over $\Sigma_{\mathtt{int}}$, and $\varphi\gamma$ is valid. Note that symbols $2, 3, \dots$ are *not* included in the signature; terms have forms like $\mathsf{sum}((1+1)+1)$.

Example 15. In [3] we see how a code snippet is translated to a $\mathbb{Z}$-TRS:

```
while (x > 0 && y > 0) {
   if (x > y) { while (x > 0) { x--; y++; } }
   else { while (y > 0) { y--; x++; } }
}
```

$$\begin{array}{lll}
\mathsf{eval}_1(x,y) \to \mathsf{eval}_2(x,y) & [x > 0 \wedge y > 0 \wedge x > y] \\
\mathsf{eval}_1(x,y) \to \mathsf{eval}_3(x,y) & [x > 0 \wedge y > 0 \wedge \neg(x > y)] \\
\mathsf{eval}_2(x,y) \to \mathsf{eval}_2(x + -1, y + 1) & [x > 0] \\
\mathsf{eval}_2(x,y) \to \mathsf{eval}_1(x,y) & [\neg(x > 0)] \\
\mathsf{eval}_3(x,y) \to \mathsf{eval}_3(x + 1, y + -1) & [y > 0] \\
\mathsf{eval}_3(x,y) \to \mathsf{eval}_1(x,y) & [\neg(y > 0)]
\end{array}$$

In fact, $\mathbb{Z}$-TRSs are very close to our LCTRSs, but with a fixed theory signature. Although there is no concept of "values", there is no harm to internally replacing ground terms over $\Sigma_{\mathtt{int}}$ by the corresponding value (in a tool, or when manually rewriting terms), because the symbols in Σ_{int} do not occur in any left-hand side.

For every $\mathbb{Z}$-TRS, we can define an LCTRS which is roughly the same, modulo calculation of integer values. We can do this as follows: let $\Sigma_{theory} := \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\} \cup \{\mathsf{true}, \mathsf{false} : \mathsf{bool},\ - : [\mathsf{int}] \Rightarrow \mathsf{int},\ +, * : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int},\ =, >, \geq : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int},\ \neg : [\mathsf{bool}] \Rightarrow \mathsf{bool},\ \wedge : [\mathsf{bool} \times \mathsf{bool}] \Rightarrow \mathsf{bool}\}$ and $\Sigma_{terms} := (\Sigma \setminus \Sigma_{\mathtt{int}}) \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\}$. Every $\mathbb{Z}$-TRS rule is already a standard rule in this LCTRS, and every term in the original $\mathbb{Z}$-TRS is still a term.

[2] The authors use simplifications of this formalism for different applications. For example, multiplication is omitted, or custom symbols must have output sort unit.

Theorem 3. *We can derive, for all ground terms s, t:*

- *if $s \to_{\mathcal{R}} t$ in a $\mathbb{Z}$-TRS, and $s \to_{\mathtt{calc}}^{*} s' \in \mathcal{T}erms(\Sigma_{terms} \cup \Sigma_{theory}, \mathcal{V})$, then exists t' such that $t \to_{\mathtt{calc}}^{*} t'$ and $s' \to_{\mathcal{R}}^{+} t'$ in the corresponding LCTRS;*
- *if $s \to_{\mathtt{rule}} t$ in the corresponding LCTRS, and $s' \in \mathcal{T}erms(\Sigma, \mathcal{V})$ such that $s' \to_{\mathtt{calc}}^{*} s$, then there is a term t' such that $t' \to_{\mathtt{calc}}^{*} t$ and $s' \to_{\mathcal{R}} t'$ in the original $\mathbb{Z}$-TRS.*

Thus, results from LCTRSs typically extend to $\mathbb{Z}$-TRSs. As with ITRSs, $\mathbb{Z}$-TRSs cannot model the behaviour of LCTRSs. Being fundamentally restricted to the integers, they cannot easily represent Example 6, nor Example 12. An extension of $\mathbb{Z}$-TRSs, which admits all integers in the signature, can model a variation of Example 6, as discussed in [6]. This analysis uses extra rules to "normalise" integers to their range, e.g. $\mathsf{loop}_2(x) \to \mathsf{loop}_3(x * 2)$, $\mathsf{loop}_3(x) \to \mathsf{loop}_3(x - 256)\ [x \geq 256]$, $\mathsf{loop}_3(x) \to \mathsf{loop}(x)\ [x \geq 0 \wedge 256 \geq x]$.

5.4 Constrained Equational Systems

For a very different direction, let us consider a system from the further past. In [11], a framework for constrained deduction is developed, which uses constrained terms and rules. Like the current paper, the interpretation of function symbols ($\mathcal{J}_f$) is not fixed, but assumed to be given by the user. There is no notion of values, however. This fits with a typical usage of the formalism, where the underlying model is the set of terms modulo some theory.

Example 16. We consider a constrained system with symbols $* : [\mathsf{term} \times \mathsf{term}] \Rightarrow \mathsf{term}$, $\mathsf{a}, \mathsf{b} : \mathsf{term}$, $=_{\mathsf{AC}}$: $[\mathsf{term} \times \mathsf{term}] \Rightarrow \mathsf{bool}$. The model $\mathcal{I}_{\mathsf{term}}$ is the set of terms over $\{*, \mathsf{a}, \mathsf{b}\}$, where a, b and $*$ are interpreted as themselves and $=_{\mathsf{AC}}$ is interpreted as equality on terms modulo AC (associativity and commutativity) of $*$.

Unlike LCTRSs, this formalism has no separate "term signature": all function symbols have a meaning in the model, and may occur in both terms and constraints. Rules have the form $s \to t\ [\varphi]$ and are used for example to simplify constrained terms (called *constrained formulas*) modulo an equational theory.

Example 17. In the signature from Example 16, we consider a rule $(x * x) * x \to \mathsf{a}\ [\neg(x =_{\mathsf{AC}} \mathsf{a})]$, which matches modulo AC. The constrained formula $x * (\mathsf{b} * (\mathsf{b} * y)) =_{\mathsf{AC}} \mathsf{a} * \mathsf{b}\ [x =_{\mathsf{AC}} \mathsf{b}]$ is AC-equivalent to $((x * \mathsf{b}) * \mathsf{b}) * y =_{\mathsf{AC}} \mathsf{a} * \mathsf{b}\ [x =_{\mathsf{AC}} \mathsf{b}]$, which can be reduced to $\mathsf{a} * y =_{\mathsf{AC}} \mathsf{a} * \mathsf{b}\ [x =_{\mathsf{AC}} \mathsf{b}]$. This notion of reduction is called *total simplification*. There is also a notion of *partial simplification*, where constrained terms are reduced to pairs. This happens when a rule does not necessarily match; for example the constrained formula $x * (\mathsf{b} * (\mathsf{b} * y)) =_{\mathsf{AC}} \mathsf{a} * \mathsf{b}\ [\neg(x =_{\mathsf{AC}} y]$ reduces to the pair $\mathsf{a} * y =_{\mathsf{AC}} \mathsf{a} * \mathsf{b}\ [\neg(x =_{\mathsf{AC}} y \wedge (x * x) * x =_{AC} (x * \mathsf{b}) * \mathsf{b} \wedge \neg(x =_{\mathsf{AC}} \mathsf{a})]$ and $((x * \mathsf{b}) * \mathsf{b}) * y =_{\mathsf{AC}} \mathsf{c} * \mathsf{b}\ [\neg(x =_{\mathsf{AC}} y) \wedge \neg((x * x) * x =_{AC} (x * \mathsf{b}) * \mathsf{b} \wedge \neg(x =_{\mathsf{AC}} \mathsf{a})]$.

There are many similarities between these equational systems and LCTRSs; to a large extent they can be seen as non-standard LCTRSs. From this perspective, complete simplification is exactly constrained rewriting as we saw in Section 4. We have no notion of partial simplification, because it fundamentally relies on the symbols from terms being moved into the constraint, but similar techniques could be defined for the special case that $\Sigma_{terms} = \emptyset$.

However, LCTRSs do not allow reasoning modulo a theory, which alters fundamental properties like computability of reduction. Moreover, the systems from [11] violate an essential rule in LCTRSs: logical terms reduce only to their value. In the presence of rules like $x + (y + z) \rightarrow y$, many analysis techniques break.

Thus, while there is an overlap in expressibility between these two formalisms, we do not claim to cover or improve on this style of constrained rewriting. The dynamics of the systems are too different, and so are their purposes: where equational systems are designed for equational reasoning in logic, LCTRSs are designed for analysing programs. In the rest of this section, we have seen how LCTRSs relate to several formalisms which share this goal.

6 Analysing LCTRSs

Several times we have alluded to the ease of analysis in LCTRSs, so it is time to give some indication of how this is done. Unfortunately, we cannot do this justice, as there are many questions for analysis and little space. To give some ideas of how common techniques extend to LCTRSs, we will now briefly study some basic confluence and termination results.

6.1 (Weak) Orthogonality

Confluence is the property that whenever $s \rightarrow_{\mathcal{R}}^* t$ and $s \rightarrow_{\mathcal{R}}^* q$ there is some w such that $t \rightarrow_{\mathcal{R}}^* w$ and $q \rightarrow_{\mathcal{R}}^* w$. We will extend the common notion of *orthogonality*, a property which implies confluence, to LCTRSs.

It is well-known that for any pair of terms which can be unified, there is a *most general unifier*. That is, if s and t have distinct variables, and $s\gamma = t\gamma$, there is some δ such that also $s\delta = t\delta$, and any unifying substitution γ can be written as $\epsilon \circ \delta$ for some ϵ (here, $(\epsilon \circ \delta)(x) = \delta(x)\epsilon$ if $x \in Dom(\delta)$ and $\epsilon(x)$ otherwise). A substitution γ *respects variables* of a rule ρ if $\gamma(x)$ is a value or variable for all x in $LVar(\rho)$. If γ respects variables of $l \rightarrow r\ [\varphi]$, then $l\gamma \rightarrow r\gamma\ [\varphi\gamma]$ is also a rule.

Definition 1 (Critical Pair). *Given rules $\rho_1 \equiv l_1 \rightarrow r_1\ [\varphi_1]$ and $\rho_2 \equiv l_2 \rightarrow r_2\ [\varphi_2]$ with distinct variables, the* critical pairs *of ρ_1, ρ_2 are all tuples $\langle s, t, \varphi \rangle$ where:*

- *l_1 can be written as $C[l_1']$, where l_1' is not a variable, but is unifiable with l_2;*
- *$C \neq \Box$, or not $\rho_1 = \rho_2$ modulo renaming of variables, or $Var(r_1) \not\subseteq Var(l_1)$;*
- *the most general unifier γ of l_1' and l_2 respects variables of both ρ_1 and ρ_2;*
- *$\varphi_1\gamma \wedge \varphi_2\gamma$ is satisfiable;*
- *$s = r_1\gamma$ and $t = (C\gamma)[r_2\gamma]$ and $\varphi = \varphi_1\gamma \wedge \varphi_2\gamma$.*

The critical pairs for calculations of a rule ρ are all critical pairs of ρ with any "rule" of the form $f(x_1, \ldots, x_n) \rightarrow y\ [y = f(\boldsymbol{x})]$ with $f \in \Sigma_{theory} \setminus \mathcal{V}al$.

Note that a rule $f \rightarrow g(x)$ has a critical pair with its own renamed copy: $\langle g(x), g(y), \mathsf{true} \wedge \mathsf{true} \rangle$. This is necessary because fresh variables in the right-hand sides of rules are a very likely source of non-confluence.

Example 18. Consider the following rules:

$$\begin{array}{lrll}
(\rho_1) & f(x_1, y_1) & \to g(x_1 + y_1) & [x_1 \geq y_1] \\
(\rho_2) & f(x_2, y_2) & \to g(x_2) & [x_2 \leq y_2] \\
(\rho_3) & f(x_3, y_3) & \to g(y_3) & [x_3 < y_3] \\
(\rho_4) & f(x_4, x_4 + y_4) & \to g(y_4) & [x_4 > \mathsf{0}]
\end{array}$$

There are no critical pairs between ρ_1 and ρ_3: although $f(x_1, y_1)$ and $f(x_3, y_3)$ can be unified (with most general unifier $[x_1 := x,\ x_3 := x,\ y_1 := y,\ y_3 := y]$), the formula $x \geq y \wedge x < y$ is not satisfiable. On the other hand, ρ_1 and ρ_2 do admit a critical pair: $\langle g(x+y), g(y), x \geq y \wedge x \leq y \rangle$. None of the rules $\rho_1,\ \rho_2$ or ρ_3 gives a critical pair with ρ_4, since in the resulting mgu γ we have $\gamma(y_1) = x + y$, and thus this substitution does not respect the variables of ρ_1, ρ_2, ρ_3. Finally, ρ_4 has a critical pair for calculations, $\langle g(y), f(x, z), x > \mathsf{0} \wedge z = x + y \rangle$.

Definition 2 (Weak Orthogonality). *A critical pair $\langle s, t, \varphi \rangle$ is* trivial *if $s\ [\varphi] \approx t\ [\varphi]$. An LCTRS $\mathcal{R}$ is* weakly orthogonal *if the left-hand side of each rule is linear (no variable occurs more than once), and for any pair $\rho_1, \rho_2 \in \mathcal{R}$: every critical pair between ρ_1 and a variable-renamed copy of ρ_2, and every critical pair of ρ_1 for calculations, is trivial. It is* orthogonal *if there are no critical pairs.*

The following result follows much like its unconstrained counterpart:

Theorem 4. *A weakly orthogonal LCTRS is confluent.*

Example 19. sum is orthogonal, so by Theorem 4 this LCTRS is confluent.

6.2 The Recursive Path Ordering

To prove termination of a TRS, it suffices to show that its rules are included in the *recursive path ordering* [2], a well-founded ordering $\succ$ which is monotonic and stable under substitutions. We will consider a simple variation of this ordering. To deal with the possibly infinite number of values, we assume that Σ_{theory} contains a symbol $\sqsupset_\iota$ for all sorts ι occurring in $\mathcal{V}al$, which is mapped to a well-founded ordering $>_\iota$ in $\mathcal{I}_\iota$. For example, we might take $\sqsupset_{\mathsf{int}} = \lambda xy.x > y \wedge x \geq 0$. We also assume given a well-founded ordering $\rhd$ on the symbols of $\Sigma_{terms} \backslash \Sigma_{theory}$.

The recursive path ordering is defined by the following derivation rules:

1. $s \succeq t\ [\varphi]$ if one of the following holds:
   (a) $s, t \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V}ar(\varphi))$, and $\varphi \Rightarrow (s = t \vee s \sqsupset t)$ is valid
   (b) $s = f(s_1, \dots, s_n), t = f(t_1, \dots, t_n)$ with $f \notin \Sigma_{theory}$ and each $s_i \succeq t_i\ [\varphi]$
   (c) $s \succ t\ [\varphi]$, or $s = t$ is a variable
2. $s \succ t\ [\varphi]$ if one of the following holds:
   (a) $s, t \in \mathcal{T}erms(\Sigma_{theory}, \mathcal{V}ar(\varphi))$, and $\varphi \Rightarrow s \sqsupset t$ is valid
   (b) $s = f(s_1, \dots, s_n)$ with $f \in \Sigma_{terms} \setminus \Sigma_{theory}$ and one of:
      i. $s_i \succeq t\ [\varphi]$ for some $i \in \{1, \dots, n\}$
      ii. $t = g(t_1, \dots, t_m)$ with $g \in \Sigma_{theory}$ or $f \rhd g$, and for all i: $s \succ t_i\ [\varphi]$
      iii. $t = f(t_1, \dots, t_n)$, all $s_i \succeq t_i\ [\varphi]$ and for at least one i: $s_i \succ t_i\ [\varphi]$
      iv. $t \in \mathcal{V}ar(\varphi)$

Theorem 5. *An LCTRS $\mathcal{R}$ is terminating if we can choose a suitable $\sqsupset_\iota$ for all ι, and some well-founded $\rhd$, such that $l \succ r\ [\varphi]$ for all $l \to r\ [\varphi] \in \mathcal{R}$.*

Proof. We can define a pair $(\equiv, >)$ of an equivalence relation and a compatible ordering with $\to_{\mathsf{calc}} \subseteq \equiv$ and $C[s] > C[t]$ if $s \succ t\ [\mathsf{true}]$ and $s \notin \mathcal{T}erms(\Sigma_{theory}, \emptyset)$. Having these, we observe first that $>$ is well-founded, and second that if $l \succeq r\ [\varphi]$, then $l\gamma \succ r\gamma\ [\mathsf{true}]$ for all substitutions γ which respect $l \to r\ [\varphi]$.

Example 20. Taking $n \sqsupset_{\mathsf{int}} m$ if $n > m$ and $n \geq 0$, the sum system is terminating by the recursive path ordering: For the first rule, $\mathsf{sum}(x) \succ \mathsf{0}\ [\mathsf{0} \geq x]$ by 2(b)ii. For the second, writing $\varphi = \neg(\mathsf{0} \geq x)$, we have $\mathsf{sum}(x) \succ x + \mathsf{sum}(x + -1)\ [\varphi]$ by 2(b)ii because $\mathsf{sum}(x) \succ x\ [\varphi]$ by 2(b)iv, and $\mathsf{sum}(x) \succ \mathsf{sum}(x + -1)\ [\varphi]$ by 2(b)iii because $x \succ x + -1\ [\varphi]$ by 2a, because $\varphi \Rightarrow (x > x + -1 \wedge x \geq \mathsf{0})$ is valid.

Note that Example 3, with encoded constraints, cannot be handled by RPO.

Of course, this is a very basic version of the recursive path ordering. There are various ways to strengthen the technique, but this is left for future work.

6.3 Observations

Both when analysing confluence and termination, a pattern appears: existing techniques extend in fairly natural way, with the constraints handled by proving validity of some formula. In other techniques we have studied but omitted here (such as dependency pairs and inductive equality proofs) a similar pattern arises.

Importantly, this pattern does not depend on the kind of theory we use: analysis takes a similar form whether we reason about integer arrays, reals or bitvectors. The difference is in how to solve the resulting formulas. When automatically analysing properties of LCTRSs, it seems natural to combine a dedicated analysis tool with SMT-solvers for the theory of interest. This way, we can immediately profit from the continuing improvement of the SMT-community, without having to adjust our methods when a new theory is explored.

7 Conclusion

In this paper, we have studied *logical constrained term rewriting systems.* LCTRSs offer an approach to program analysis for a large variety of languages and analysis questions. Due to their similarity to normal term rewriting, we can easily transpose the many powerful techniques of traditional term rewriting. However, by natively handling constraints, we obtain a much simpler analysis than if we were to encode the constraints in the rules.

In conclusion, LCTRS can be summarised with four keywords: They are *natural*: values in the logic are modelled with constants, and calculations do not need to be encoded. They are *general*: LCTRSs are not restricted to for instance the integers, but can handle all kinds of theories. They are *versatile*: LCTRSs can model a wide range of problems, from termination and overflow analysis to program equivalence, and can represent examples from many existing formalisms of constrained or integer rewriting. Finally, they are *flexible*: common analysis techniques for term rewriting extend to LCTRSs without much effort.

In the future, we aim to provide a tool to rewrite and analyse LCTRSs. Such analysis would not necessarily need special treatment for the various theories: in many cases (as we saw in Section 6), an LCTRS problem can be converted into a sequence of SMT-queries which might be fed into an external solver.

In addition, we hope to extend translations of program analysis from e.g. [3,9,13] with arrays and bitvectors, thus making use of the greater generality of LCTRSs, and the power of SMT-solvers for various theories.

References

1. Community. SMT-LIB, `http://www.smtlib.org/`
2. Dershowitz, N.: Orderings for term rewriting systems. Theor. Comput. Sci. 17(3), 279–301 (1982)
3. Falke, S., Kapur, D.: A term rewriting approach to the automated termination analysis of imperative programs. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 277–293. Springer, Heidelberg (2009)
4. Falke, S., Kapur, D.: Rewriting induction + linear arithmetic = decision procedure. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 241–255. Springer, Heidelberg (2012)
5. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Schmidt-Schauß, M. (ed.) Proc. RTA 2011. LIPIcs, vol. 10, pp. 41–50. Dagstuhl (2011)
6. Falke, S., Kapur, D., Sinz, C.: Termination analysis of imperative programs using bitvector arithmetic. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 261–277. Springer, Heidelberg (2012)
7. Fuhs, C., Giesl, J., Parting, M., Schneider-Kamp, P., Swiderski, S.: Proving Termination by Dependency Pairs and Inductive Theorem Proving. Journal of Automated Reasoning 47(2), 133–160 (2011)
8. Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P., Falke, S.: Proving termination of integer term rewriting. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 32–47. Springer, Heidelberg (2009)
9. Furuichi, Y., Nishida, N., Sakai, M., Kusakari, K., Sakabe, T.: Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. IPSJ Trans. Program. 1(2), 100–121 (2008)
10. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for Haskell by term rewriting. ACM Transactions on Programming Languages and Systems 33(2), 7:1–7:39 (2011)
11. Kirchner, C., Kirchner, H., Rusinowitch, M.: Deduction with symbolic constraints. Revue Française d'Intelligence Artificielle 4(3), 9–52 (1990)
12. Nakabayashi, N., Nishida, N., Kusakari, K., Sakabe, T., Sakai, M.: Lemma generation method in rewriting induction for constrained term rewriting systems. Computer Software 28(1), 173–189 (2010) (in Japanese)
13. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of java bytecode by term rewriting. In: Lynch, C. (ed.) Proc. RTA 2010. LIPIcs, vol. 6, pp. 259–276. Dagstuhl (2010)
14. Sakata, T., Nishida, N., Sakabe, T.: On proving termination of constrained term rewrite systems by eliminating edges from dependency graphs. In: Kuchen, H. (ed.) WFLP 2011. LNCS, vol. 6816, pp. 138–155. Springer, Heidelberg (2011)
15. Schneider-Kamp, P., Giesl, J., Ströder, T., Serebrenik, A., Thiemann, R.: Automated termination analysis for logic programs with cut. Theory and Practice of Logic Programming 10(4-6), 365–381 (2010)

Author Index