Adenilso Simao
Carroll Morgan (Eds.)

LNCS 7021

Formal Methods: Foundations and Applications

14th Brazilian Symposium, SBMF 2011
São Paulo, Brazil, September 2011
Revised Selected Papers

Springer

Adenilso Simao   Carroll Morgan (Eds.)

Formal Methods: Foundations and Applications

14th Brazilian Symposium, SBMF 2011
São Paulo, Brazil, September 26-30, 2011
Revised Selected Papers

Springer

Volume Editors

Adenilso Simao
University of São Paulo
Institute of Mathematics and Computer Science
Avenida Trabalhador, são-carlense, 400
Zip Code 13566-590, São Carlos, SP, Brazil
E-mail: adenilso@icmc.usp.br

Carroll Morgan
University of New South Wales
School of Computer Science and Engineering
Sydney, NSW 2052, Australia
E-mail: carrollm@cse.unsw.edu.au

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-25031-6 e-ISBN 978-3-642-25032-3
DOI 10.1007/978-3-642-25032-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011939535

CR Subject Classification (1998): D.2.4, D.2, F.3, D.3, D.1, K.6, F.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

Printed on acid-free paper

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

Preface

This volume contains the papers presented at SBMF 2011: the 14th Brazilian Symposium on Formal Methods. The conference was held in the city of São Paulo, Brazil, colocated with CBSoft 2011, the Second Brazilian Conference on Software: Theory and Practice.

The conference program included two invited talks, given by Catuscia Palamidessi (INRIA Saclay, France) and John Harrison (Intel Corporation, USA).

A total of 13 research papers were presented at the conference and are included in this volume; they were selected from 37 submissions. The submissions came from 12 countries: Algeria, Brazil, China, Finland, France, Germany, India, Ireland, Malaysia, Portugal, Spain, the UK and the USA. There were also sessions specially devoted to a track of short papers, and these are published separately as a technical report including papers describing work in progress.

The deliberations of the Program Committee and the preparation of these proceedings were handled by EasyChair, which indeed made our lives much easier.

We are grateful to the Program Committee, and the additional reviewers, for their hard work in evaluating submissions and suggesting improvements. SBMF 2011 was organized by *Escola de Artes, Ciências e Humanidades da Universidade de São Paulo* (EACH/USP) and *Faculdade de Computação e Informática da Universidade Presbiteriana Mackenzie* (FCI/Mackenzie) under the auspices of the Brazilian Computer Society (SBC). We are very thankful to the organizers of this year's conference, Fatima L.S. Nunes Marques (EACH/USP) and Ana Cristina Vieira de Melo (IME/USP), and we are specially thankful to CBSoft 2011's organizers Marcelo Fantinato (EACH/USP) and Luciano Silva (FCI/Mackenzie), who arranged everything and made the conference run smoothly.

The conference was sponsored by the following organizations, which we thank for their generous support:

- CNPq, the Brazilian Scientific and Technological Research Council
- CAPES, the Brazilian Higher Education Funding Council
- FAPESP, São Paulo Research Foundation
- Google Inc.
- Universidade de São Paulo
- Universidade Presbiteriana Mackenzie

August 2011

Adenilso Simao
Carroll Morgan

Conference Organization

Program Chairs

Adenilso Simao	ICMC/USP, Brazil
Carroll Morgan	UNSW, Australia

Steering Committee

Adenilso Simao	ICMC-USP, Brazil
Carroll Morgan	UNSW, Australia
David Naumann	Stevens Institute of Technology, USA
Jim Davies	University of Oxford, UK (Co-chair)
Jim Woodcock	University of York, UK
Leila Silva	UFS, Brazil
Marcel Oliveira	UFRN, Brazil (Co-chair)
Patrícia Machado	UFCG, Brazil
Rohit Gheyi	UFCG, Brazil

Program Committee

Aline Andrade	UFBA, Brazil
David Aspinall	University of Edinburgh, UK
Luis Barbosa	Universidade do Minho, Portugal
Michael Butler	University of Southampton, UK
Andrew Butterfield	Trinity College Dublin, Ireland
Ana Cavalcanti	University of York, UK
Marcio Cornelio	UFPE, Brazil
Andrea Corradini	Università di Pisa, Italy
Jim Davies	University of Oxford, UK
David Deharbe	UFRN, Brazil
Ewen Denney	RIACS/NASA, USA
Clare Dixon	University of Liverpool, UK
Jorge Figueiredo	UFCG, Brazil
Rohit Gheyi	UFCG, Brazil
Rolf Hennicker	Ludwig-Maximilians-Universität München, Germany
Juliano Iyoda	UFPE, Brazil
Gerald Luettgen	University of Bamberg, Germany
Patrícia Machado	UFCG, Brazil

Ana Melo	USP, Brazil
Stephan Merz	INRIA Lorraine, France
Anamaria Moreira	UFRN, Brazil
Alvaro Moreira	UFRGS, Brazil
Arnaldo Moura	Unicamp, Brazil
Alexandre Mota	UFPE, Brazil
David Naumann	Stevens Institute of Technology, USA
Daltro Nunes	UFRGS, Brazil
Jose Oliveira	Universidade do Minho, Portugal
Marcel Oliveira	UFRN, Brazil
Alberto Pardo	Universidad de la Republica, Uruguay
Alexandre Petrenko	CRIM, Canada
Leila Ribeiro	UFRGS, Brazil
Augusto Sampaio	UFPE, Brazil
Leila Silva	UFSE, Brazil
Heike Wehrheim	University of Paderborn, Germany
Jim Woodcock	University of York, UK

External Reviewers

Ludwig Adam
Sebastian Bauer
Florent Bouchy
Flavio S Correa Da Silva
Alexander Ditter
Arnaud Dury
Andrew Edmunds
Christoph Gladisch
Alexander Knapp
Giovanny Lucero
Kenneth Mackenzie
Tiago Massoni
Iain Mcginniss
Jan Tobias Muehlberg
Ingrid Nunes
Stefan Rieger
Paulo Salem Da Silva
Luis Sierra
Renato Alexandre Silva
Colin Snook
Daniele Nantes Sobrinho
Dominik Steenken

Local Organization

Marcelo Fantinato	EACH/USP (General Chair)
Luciano Silva	FCI/Mackenzie (General Chair)
Fátima L.S. Nunes Marques	EACH/USP
Ana Cristina Vieira de Melo	IME/USP

Table of Contents

Model Transformation and Induced Instance Migration: A Universal Framework

Harald König, Michael Löwe, and Christoph Schulz

University of Applied Sciences, FHDW Hannover,
Freundallee 15, 30173 Hannover, Deutschland
{harald.koenig,michael.loewe,christoph.schulz}@fhdw.de

Abstract. Software restructuring and refactoring facilitate the use of models as primary artifacts. Model evolution becomes agile if consistency between evolving models and depending artifacts is spontaneously maintained. In this paper we study endogenous model transformations at medium or fine granularity with impact on data structures and objects. We propose a formal framework in which transformation rules for class models can be formulated, whose application induces automatic migration of corresponding data structures. The main contribution is a correctness criterion for rule-induced instance migration based on initial semantics.

Keywords: Agility, Model Transformation, Refactoring, Data Migration, Initial Semantics.

1 Introduction

The taxonomy of software evolution [20] identifies exogenous and endogenous model transformations, dependent on whether meta-models differ or not. The proposed categorization distinguishes (amongst others) between temporal properties (when and how often changes are performed), object of change (the artifacts, their granularity, and the impact of change), and change support (degree of automation and formality as well as the change type). The subject of this paper are *endogenous* model transformations at medium or fine granularity, i.e. changes at the level of class structures or attribute assignments. We abstract from the temporal aspects, i.e. changes may occur e.g. during run-time or load-time of a software. The dominant research aspect, however, is the change impact on existing objects[1] that are typed in changing models.

Models and instances are equally structured: A model M is a graph (a diagram where nodes are classes and edges are associations between them), an instance I is also a graph which depicts object structures, e.g. business entities probably distributed over several database tables, or object parts together with their linking structure. From the theory of *algebraic graph transformations* [6] we adapt the idea of specifying structural transformations with the help of graph

[1] The terms *class* and *object* will always be used in the object-oriented jargon.

homomorphisms. Moreover, we pick up the paradigma of using morphisms from source to target model for adding objects and morphisms from target to source model for deleting. Both types of morphisms are combined to form *transformation rules*, that appear as spans

$$M \xleftarrow{l} K \xrightarrow{r} N \tag{1}$$

where M is the source and N is the target model of the transformation. If l and r are injective, the intermediate model K describes the preserved context of the transformation. However, to enable true relations between source and target, we allow the framework to utilize non-monic arrows l and r which serve as specifications for unfolding and merging. Examples for unfolding and merging are the patterns "Extract Class" and "Inline Class" in [9]. In this case, K will transport structures that shall be moved in the altering model. This will be demonstrated in the introductory example in Section 2.

We want to emphasize the differences of our approach to the use of *graph transformations* for model transformations. Following [19], endogenous transformations are specified on the meta-model level (M2 in the meta-architecture of MOF[2]) and can then be applied locally to affected parts of a data model or class model (M1) using a double pushout (DPO) construction. Applied to models (M1) and data (M0), this would mean to adjust affected object structures by matching class structures to it. This requires to carry out many local applications of a transformation rule. This "multi-matching" process can formally be described by amalgamated graph transformations [26], but is still limited to monomorphic transformation rules. Non-injective rule morphisms are admitted in [3,16], but these approaches are not equipped with "multi-matching" methods.

Consequently, we propose a technique which automatically matches affected instances all at once. This universal quantification is achieved by changing the universe of discourse: Instead of considering a graph I of objects that are indexed by types of the model[3], we prefer to think in terms of typing morphisms $I \xrightarrow{t} M$, i.e. fibres over the model[4], such that the *complete* typing morphism can become part of a *single* transformation procedure. It is well-known that interrelations between models in the indexed view yield forgetful functors, and that they get traded for pullbacks in the fibred view [8]. Thus, the forthcoming approach involves pullbacks if a rule (1) is applied to an instance t, e.g. the pullback of l and t.

The objectives of this paper are threefold:

- To present a method for *automatic* and *freely generated* migration of object structures in fibered contexts due to adjustments of their class models;
- To put conditions in place that guarantee correct execution of this method;
- To define correctness based on an underlying mathematical formalism.

[2] Meta Object Facility.
[3] The classical viewpoint in the theory of algebraic specifications.
[4] The practioner's viewpoint in software engineering.

The results shall serve as preparation for the development of suitable tools to perform automatic migration of existing data if a system designer has completed a model refactoring or even a semantic-changing model evolution. The work of [25] is extended by stating a criterion which can tell a designer in advance if an automatic migration is possible or whether semi-automatic actions have to be considered. The decision depends on the way object structures are restricted due to cardinalities or similar constraints. The second novelty of the present paper is a functor-based underpinning of migration semantics in different categories.

The main definitions (*Migration, Correctness*) and the main theorem (Theorem 2 on the *Existence of unique correct migrations*) are presented in Section 3. Section 4 defines a suitable category in which correctness of the construction of the introductory example is verified and investigates a different scenery (with a deviating category based on an implicational specification). Finally, we state related work in Section 5 and sketch directions for future research in Section 6.

2 An Introductory Example

A typical class model refactoring action is the move of an attribute from class A to class B. This is sometimes reasonable if A and B are related via inheritance, aggregation, or some other tight relationship. Similar actions are discussed in [9] ("Move Field", "Pull up Field"). Suppose in a source model M, persons possess several email addresses and each person may reference an object which contains further contact information, cf. the left model in Figure 1. An email, however,

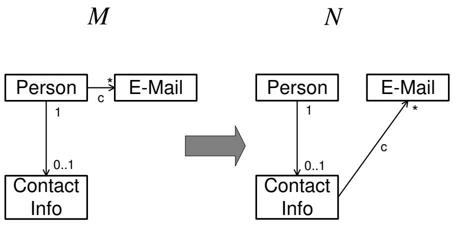


Fig. 1. Refactoring Rule "Move Attribute"

is a contact information, so the designer of the application decides to correct this defect by moving the source of attribute c from *Person* to *Contact Info*. This yields a target model N. For objects of type *Person* to be migrated, one requires that (1) sources of links to email addresses shall be shifted accordingly, (2) object structure shall remain the same if there was no email address, and (3) in the case where an email was the only contact, a new *Contact Info*-object shall be created, cf. Figure 2. If we want to specify these instance migrations with the help of graph transformations, we have to define several rules, negative application conditions, and amalgamations. In contrast to that, our goal is to

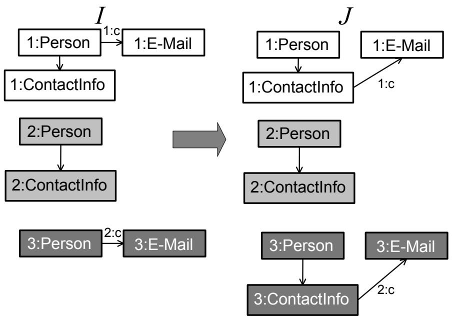


Fig. 2. Moving Links

settle the transformation with *only one* rule. For this, we interpret instances and models as graphs $G = (V, E, src, tgt : E \to V)$. Moreover, we assume that in these graphs, each node x is equipped with an "identity edge" id_x (i.e. $src(id_x) = x = tgt(id_x)$). This simulates "mapping of edges to nodes" by mapping edges to identity edges.

$$\begin{tikzcd}
M & K \arrow[l, "l"'] \arrow[r, "r"] & N \\
I \arrow[u, "t_1"] \arrow[ur, phantom, "(PB)"] & H \arrow[l, "l'"] \arrow[r, "r'"'] \arrow[u, "t"'] & J \arrow[u, "t_2"']
\end{tikzcd}$$

Fig. 3. Instance Migration $t_1 \mapsto t_2$

The novelty of the presented approach is the transformation of *complete* typing morphisms $I \xrightarrow{t_1} M$ by means of a combination of pullback (PB) and quotient construction, see Figure 3. I.e. we first construct the pullback of t_1 and l. Then the right hand square is constructed with the help of a surjective quotient map r', its kernel being a subrelation of the kernel of $r \circ t$. We postpone the details of this construction to Section 4. The resulting typing morphism t_2 is the unique arrow due to the homomorphism theorem. This yields two commuting squares as in Figure 3.

We will demonstrate these two steps in the example introduced above: Let M and N be source and target model as in Figure 1. In order to determine a complete span $M \xleftarrow{l} K \xrightarrow{r} N$, a thought-out definition of the guiding intermediate model K has to be found. For this, let us abbreviate $A :=$ *Person*, $B :=$ *Contact Info*, and $C :=$ *E-Mail*. In a typing morphism $I \longrightarrow M$ nodes and edges of M will be called *model elements*, nodes and edges of I are called *instance elements*.

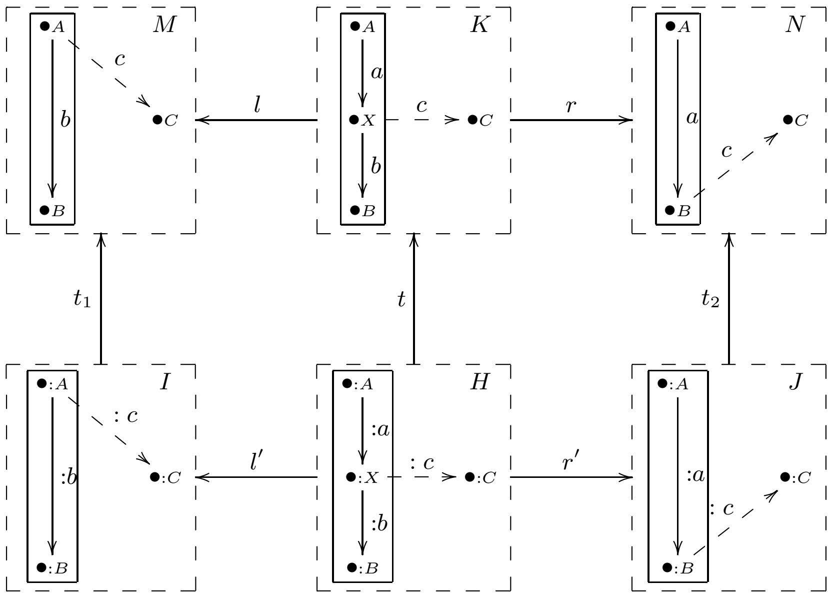


Fig. 4. Rule Application (identity edges are not shown)

For the definition of K, it is obvious that deletion of c ($c \in M - l(K)$) and adding (another) c ($c \in N - r(K)$) as in a graph transformation rule is not adequate anymore. The reason for this is that adding a model element need not necessarily infer adding a correspondingly typed instance element[5]. Thus, we need to have a preimage of c in K. But if this preimage has source A in graph K, it is mapped to an edge in N with source $r(A)$, which means that its source is not moved. Moreover, if it has source B in K, it can only origin from A in M if $l(A) = l(B)$ which is not desired. Thus, the only solution is to introduce an auxiliary node X in K. Then the rule can be specified as in the upper part of Figure 4 in which l maps the elements according to their names except $l(X) = A$ and $l(a) = id_A$ (identity edges are not shown in the graphs). This allows the attribute c to move "half-way" down. The move is completed in the right-hand side of the rule by means of the assignments $r(X) = B$ and $r(b) = id_B$.

For reasons of clarity, we first demonstrate the application of the rule only for a single object of type A: In the lower part of Figure 4, :A possesses contact information (:B) and email address (:C). Pulling back the typed instance t_1 along l causes unfolding of :A, the result being :A and :X together with an a-typed link between them. Additionally, the source of :c is moved to :X. Since $r(X) = r(B)$, in the second step, the artificial :X is merged with :B by the quotient map r', such that the link's source moves on to :B. We obtain a migrated instance

[5] A newly added class creates no objects of this class. Similarly, a newly added attribute should not always create corresponding links.

$J \xrightarrow{t_2} N$ and a diagram with two commuting squares as desired. Figure 5 shows the result of a different application: In this example, $:A$ does not possess a B-object. Because no $:B$ was present, the artificial $:X$ is only retyped (i.e. $:B := r'(:X)$). The overall effect, however, is the creation of a *new* contact information which now contains the email address as required.[6] In each case the intermediate typing t served as container to store moving structures (links of type c).

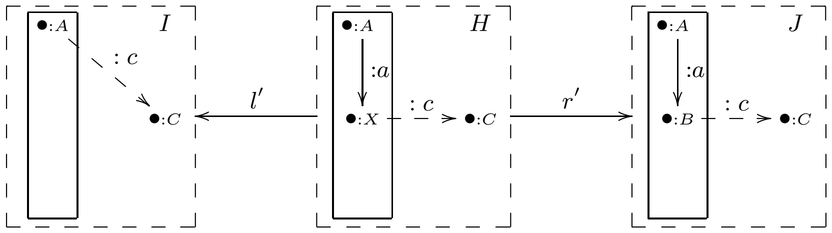


Fig. 5. Another Rule Application

In the following section, we show how to apply this technique to a whole set of objects (like in Figure 2) all at once. A general quotient construction has to take care of the interrelation of merged classes in the models: If objects of two classes A and B are in a tight relationship (e.g. if there is an association from A to B with cardinality constraints such that each A-object references at most one B-object and vice versa), which is depicted by solid association arrows[7] in Figure 4, merging due to quotient maps is admissable. If there is only a loose relationship (dashed arrows), merging may be forbidden. We formalise this issue in the following sections.

3 Migration Framework

In the sequel we require basic knowledge in category theory. We will work with categories $\mathcal{C}$, that have all limits. $Ob_{\mathcal{C}}$, $Mor_{\mathcal{C}}$ describe objects, morphisms of $\mathcal{C}$ resp. and, more precisely, for some $A, B \in Ob_{\mathcal{C}}$ the set of all arrows from A to B will be depicted by $Mor_{\mathcal{C}}(A, B)$. A subcategory $\mathcal{U}$ of $\mathcal{C}$ is called *full* if for all $A, B \in Ob_{\mathcal{U}}$ the sets $Mor_{\mathcal{U}}(A, B)$ and $Mor_{\mathcal{C}}(A, B)$ are equal. Any full subcategory is equipped with an inclusion functor $\mathscr{I} : \mathcal{U} \to \mathcal{C}$. For a fixed $M \in Ob_{\mathcal{C}}$, the comma category $\mathcal{C}{\downarrow}M$ of arrows $I \longrightarrow M$ represents M-typed instances. In the following, functors between categories are denoted by uppercase script letters.

[6] Moreover, this technique also migrates 2 :*Person* as required in Figure 2.
[7] The meaning of the solid frames in the figures will be explained during the following sections.

We recall basic facts from category theory: For $M, K \in Ob_{\mathcal{C}}$, a fixed $l \in Mor_{\mathcal{C}}(K, M)$, and any arrow $I \xrightarrow{i} M$, the pullback $P \xrightarrow{\mathscr{P}_l\, i} K$, $P \xrightarrow{l'} I$ of (l, i) is a functor $\mathscr{P}_l : \mathcal{C}{\downarrow}M \to \mathcal{C}{\downarrow}K$. If $K, N \in Ob_{\mathcal{C}}$ and $r \in Mor_{\mathcal{C}}(K, N)$ the "composing-with"-functor, which maps an arrow $j_1 \xrightarrow{\alpha} j_2$ of $\mathcal{C}{\downarrow}K$ to the arrow $r \circ j_1 \xrightarrow{\alpha} r \circ j_2$ of $\mathcal{C}{\downarrow}M$, is the left-adjoint of $\mathscr{P}_r : \mathcal{C}{\downarrow}N \to \mathcal{C}{\downarrow}K$, written $\mathscr{F}_r \dashv \mathscr{P}_r$.

Definition 1 (Migration Functor). *Let $M, N \in Ob_{\mathcal{C}}$. A functor*

$$\mathscr{M} : \mathcal{C}{\downarrow}M \to \mathcal{C}{\downarrow}N$$

is called a migration functor *from (source model) M to (target model) N.*

Definition 2 (Transformation Rule). *Let $M, K, N \in Ob_{\mathcal{C}}$. A span*

$$\mathfrak{T}\mathfrak{R} = \begin{tikzcd} M & K \arrow[l, "l"'] \arrow[r, "r"] & N \end{tikzcd}$$

is called a transformation rule*.*

In the sequel, all examples of migration functors $\mathscr{M}$ are such that $\mathscr{M} = \mathscr{M}_2 \circ \mathscr{M}_1$ where $\mathscr{M}_1 : \mathcal{C}{\downarrow}M \to \mathcal{C}{\downarrow}K$ and $\mathscr{M}_2 : \mathcal{C}{\downarrow}K \to \mathcal{C}{\downarrow}N$. As demonstrated in Section 2, $\mathscr{M}_1$ and $\mathscr{M}_2$ are required to represent pullback and quotient construction, resp.

For example, if $\emptyset$ is initial in $\mathcal{C}$, we can define a functor $\mathscr{M}_1 : \mathcal{C}{\downarrow}M \to \mathcal{C}{\downarrow}K$ by $\mathscr{M}_1 i = (\ \emptyset \longrightarrow K\ )$, mapping each arrow to the identity $id_{\emptyset}$. It can easily be checked that this yields commuting squares, but in general no pullbacks. In practice, this part of the migration deletes all instance elements of the source model. It will be considered incorrect, whereas $\mathscr{M}_1 := \mathscr{P}_l$ will be considered correct (see Definition 3), because it deletes exactly those instance elements whose class has been removed in the rule.

Examples for the second step are the composition-functor $\mathscr{F}_r$, but also the functor $\mathscr{M}_2$ mapping each $t \in \mathcal{C}{\downarrow}K$ to $N \rightarrowtail\!\!\xrightarrow{id}\!\!\twoheadrightarrow N$ in $\mathcal{C}{\downarrow}N$ taking $r' = r \circ t$. Again, we can ask for correctness: E.g. $\mathscr{M}_2$ transforms t to an instance of model N, the result being just a copy of N. This behaviour should be marked "incorrect" in general because it may not reflect the rule's properties.

If $\mathscr{M}$ is governed by a transformation rule $\mathfrak{T}\mathfrak{R}$, we call $\mathscr{M}$ *rule-induced.* As pointed out above, correctness in the first step of a rule-induced migration is achieved if $\mathscr{M}_1$ is the pullback functor. The second step will be considered correct if no information is lost due to the merge of instance structures – except if it is enforced by the transformation rule – (no confusion) and such that no unnecessary instance elements are created (no junk). These are the typical requirements of *initial semantics.* Following the semantics of parametrized data types [7], it is reasonable to search for left-adjoints of $\mathscr{P}_r$ (the counterpart of the forgetful functor along r in the indexed case).

If a model M imposes further restrictions (e.g. cardinalities or inheritance structures) on an M-typed instance I, one has to distinguish between valid and invalid arrows $I \xrightarrow{type} M$ (cf. Definition 4 in Section 4). To enable intelligent migration functors which respect these restrictions, (proper) *subcategories* have to be introduced. Together with the aforementioned initiality considerations, we propose the following view on correctness:

Definition 3 (Correctness). *Let*

$$\mathfrak{TR} = \begin{tikzcd} M & K \arrow[l, "l"'] \arrow[r, "r"] & N \end{tikzcd}$$

be a transformation rule, and $\mathcal{U}_M$, $\mathcal{U}_N$ be full subcategories of $\mathcal{C}{\downarrow}M$, $\mathcal{C}{\downarrow}N$ with corresponding inclusion functors $\mathscr{I}_M : \mathcal{U}_M \to \mathcal{C}{\downarrow}M$ and $\mathscr{I}_N : \mathcal{U}_N \to \mathcal{C}{\downarrow}N$.

A migration functor

$$\mathscr{M} : \mathcal{U}_M \to \mathcal{U}_N$$

is said to be correct *w.r.t. $\mathfrak{TR}$, if $\mathscr{M} = \mathscr{M}_2 \circ \mathscr{M}_1$ and*

1. *$\mathscr{M}_1 = \mathscr{P}_l \circ \mathscr{I}_M : \mathcal{U}_M \to \mathcal{C}{\downarrow}K$ and*
2. *$\mathscr{M}_2 \dashv \mathscr{P}_r \circ \mathscr{I}_N : \mathcal{U}_N \to \mathcal{C}{\downarrow}K$,*

cf. Figure 3 with $t = \mathscr{P}_l t_1$ and $\mathscr{M} t_1 = \mathscr{M}_2 t = t_2$.[8]

Condition 1 can always be fulfilled because $\mathcal{C}$ has all limits. But condition 2 might be violated for each choice of $\mathscr{M}_2$.

We aim at a very general setting in which a migration functor $\mathscr{M}$ is left-adjoint (i.e. a free construction), because it is well-known that many important properties are preserved by left-adjoint functors, see the remark on the preservation of co-limits in Section 6. Because typed instances not only occur in simple graph structures, but also in more general settings (e.g. typed graphs, typed attributed graphs [6] or generalized sketches [5]), we conclude that topoi[9] are the most general setting which serves our purposes. All investigated and above mentioned examples are topoi.

Indeed, if the underlying category is a topos and if $\mathscr{M}$ can be extended to $\mathcal{C}{\downarrow}M$[10], we have

Theorem 1. *In a topos $\mathcal{C}$ any correct $\mathscr{M} : \mathcal{C}{\downarrow}M \to \mathcal{U}_N$ is left-adjoint.*

Proof. By the well-known fundamental theorem of topos theory [10], $\mathcal{C}{\downarrow}M$ and $\mathcal{C}{\downarrow}K$ are topoi, too. Moreover, the pullback functor has a right-adjoint in topoi. Because left-adjoints compose, $\mathscr{M}$ is left-adjoint. □

The remainder of this section is devoted to Theorem 2 which states sufficient conditions for existence and uniqueness of correct migrations. It will be applied in Section 4. The proof of the theorem is based on the following Proposition

[8] Since these subcategories shall limit the degree of freedom of instance typings only, we do not propose to restrict the transformation rule morphisms l, r.

[9] Topoi are cartesian closed categories with a representable subobject functor. E.g. algebraically specified categories with unary operation symbols only are topoi.

[10] This is no serious restriction, see Section 2.

1 (cf. [1], Corollary 16.9 and Proposition 12.5)[11]. Recall that a monomorphism $A \overset{f}{\rightarrowtail} B$ is called *extremal* if $f = m \circ e$ with epimorphism e implies that e is an isomorphism. In this case (A, f) is called an *extremal subobject* of B.

Proposition 1. *Let $\mathcal{D}$ be a complete category and $\mathcal{U}$ be a full subcategory of $\mathcal{D}$. Then the inclusion functor $\mathscr{I} : \mathcal{U} \to \mathcal{D}$ has a left-adjoint with epimorphic unit iff $\mathcal{U}$ is closed under the formation of products and extremal subobjects.* □

Note that subcategories of algebraically specified categories are closed under the formation of (extremal) subobjects and products if and only if they can be specified in syntactic terms (i.e. by means of a set of implications with a possibly infinite number of premises).

Theorem 2. *Let $\mathfrak{T}\mathfrak{R} = M \xleftarrow{l} K \xrightarrow{r} N$ be a transformation rule in a complete category $\mathcal{C}$ and $\mathcal{U}_M, \mathcal{U}_N$ be as in Definition 3. If $\mathcal{U}_N$ is closed under the formation of products and extremal subobjects then there exists a correct migration functor w.r.t. $\mathfrak{T}\mathfrak{R}$. It is essentially unique*[12].

Proof. Let $\mathscr{I}_M : \mathcal{U}_M \to \mathcal{C} \downarrow M$ and $\mathscr{I}_N : \mathcal{U}_N \to \mathcal{C} \downarrow N$ be the participating inclusion functors. We define

$$\mathscr{M}_1 := \mathscr{P}_l \circ \mathscr{I}_M.$$

It is well-known that $\mathcal{C}{\downarrow}N$ is complete, as well. Thus, by Propositon 1 applied to $\mathcal{D} = \mathcal{C}{\downarrow}N$, there is a functor $\mathscr{F}_N : \mathcal{C}{\downarrow}N \to \mathcal{U}_N$ such that $\mathscr{F}_N \dashv \mathscr{I}_N$. Since left-adjointness is closed under composition,

$$\mathscr{M}_2 := \mathscr{F}_N \circ \mathscr{F}_r \dashv \mathscr{P}_r \circ \mathscr{I}_N : \mathcal{U}_N \to \mathcal{C}{\downarrow}K$$

guarantees correctness. Moreover, the choice of left-adjoint functors and of pullbacks is unique up to isomorphism, such that $\mathscr{M}_2 \circ \mathscr{M}_1$ is essentially unique. □

4 Application of the Framework

This section presents two different examples for the underlying category from Section 3. Both definitions are based on graphs with identity edges, i.e. the category of all algebras specified by the signature *Graph* with sorts E and V and operation symbols $src, tgt \colon E \to V$, $id \colon V \to E$.[13] We write id_x instead of $id(x)$. Let the specification Σ be equal to *Graph* together with the equations

$$\forall x \in V : src(id_x) = x \text{ and } tgt(id_x) = x.$$

The equationally specified category $\mathcal{C} := SET^{\Sigma}$ of all these graphs with graph homomorphisms between them is a full subcategory of SET^{Graph}. Both are complete (for the completeness of $\mathcal{C}$ use Birkhoff's Variety Theorem). All models and instances in the following considerations are graphs of $\mathcal{C}$.

[11] This result is applicable if the categories under considerations fulfil certain smallness assumptions which are always guaranteed in practical settings.

[12] I.e. unique up to natural isomorphism.

[13] The identity edge may be thought of a special reference to the same object (e.g. *this*-reference in Java).

4.1 Models with Component Structure

We sketch the results of [17]: Since the first migration step of our transformations is managed by the well-understood pullback construction, we focus on problems and results that are concerned with the more difficult second step of the migration. As we have already seen in Section 2, the second step of the migration may identify some elements of the intermediate instance H, cf. Figure 4. In general, however, unconditional identification is impractical. For suppose a transformation rule specifies merging two classes A and B (i.e. $r(A) = r(B)$) that have a multi-valued association between them. If an instance $:A$ is linked to two B-objects $1{:}B$ and $2{:}B$, and the rule would enforce the merge of all connected object pairs $(:A, :B)$, one would observe merging of $1{:}B$ and $2{:}B$ as well, which was not required.

Thus, we have to distinguish between "areas" in class models which permit merging and areas which don't. A good candidate for this is cardinality constraints: We will differentiate between *tight* and *loose* associations. Tight associations are of type $n{:}m$, where $n, m \in \{0..1, 1\}$. Solid arrows depicted these associations in Section 2. Dashed arrows depict loose (i.e not tight) associations. Links are depicted correspondingly, depending on how they are typed.

Let G be any class model. We say that classes A and B are *tightly associated* if there is a tight association in some direction between them. The reflexive and transitive closure of this relation can be coded by a homomorphism $g : G \to \underline{G}$ via $ker(g)$. For y a node in $\underline{G}$, the union of the equivalence classes $g^{-1}(\{y\})$ and $g^{-1}(\{id_y\})$ is called a *component* of the model graph. A reasonable component structure of areas which permit merging is generated if g maps connected tight associations to id_y for some node $y \in \underline{G}$.

If C is such a component of model K and $H \xrightarrow{t} K$ is a typed instance, the set $t^{-1}(C)$ can be partitioned into disconnected[14] parts of H. If this is done for all components of K, the instance graph is partitioned as well. In Figure 4 components are solid frames (components with exactly one element have no frame). Their preimages under the typing morphisms are framed in the same way. In this way, typing morphisms respect the component structure. If graphs H and K are equipped with component structures $h : H \to \underline{H}$ and $k : K \to \underline{K}$, this compatibility condition can be expressed by claiming that t is accompanied by a "component morphism" $\underline{t} : \underline{H} \to \underline{K}$, s.th. $(t, \underline{t}) \in Mor_{\mathcal{C}^2}(h, k)$, i.e. $\underline{t} \circ h = k \circ t$. Thus, the arrow category $\mathcal{C}^2$ is a good choice for generalising graphs that are equipped with a component structure.

We use the following conventions: Whenever a model graph or an instance graph g is considered, we assume g to have domain G (the graph under consideration) and codomain $\underline{G}$ (the components, i.e. the equivalence classes induced by g). If both components of a morphism in the arrow category have to be distinguished, we write $(t, \underline{t})$. Otherwise, we omit the second component and write just t.

[14] Not connected via links in either direction.

The choice of subcategory (cf. Definition 3) is now driven by the fact that only if a component C of H is mapped *injectively* by t, merging within the component due to a given rule is unique:

Definition 4 (Subcategory of Valid Typings). *Let $n : N \to \underline{N}$ be a model graph with component structure and $j \xrightarrow{t} n$ be a typed instance*[15]*. We call t a* valid typing*, if for all*[16] $x, y \in J$

$$j(x) = j(y) \wedge t(x) = t(y) \implies x = y \tag{2}$$

The full subcategory $\mathcal{U}$ of all valid typings of $\mathcal{C}^2 \downarrow n$ is called the subcategory of valid typings *of n.*

A tool can detect this behaviour by determining components C in H as disconnected t-preimages of components in K that depict a non-cyclic path of tightly associated classes. In [13] we proved $\mathcal{U}$ to fulfil the prerequisites of Theorem 2. Completeness of $\mathcal{C}$ implies completeness of $\mathcal{C}^2$ (by pointwise limit construction). Since the pullback functor always acts on the whole comma category and $\mathcal{C}^2$ is a topos [18], the prerequisites of Theorem 1 are also met:

Corollary 1. *Let $\mathfrak{TR} = m \xleftarrow{l} k \xrightarrow{r} n$ be a transformation rule in the category of graphs with a component structure and $\mathcal{U}$ be the subcategory of valid typings of n. Then there exists a correct, essentially unique $\mathfrak{TR}$-induced migration functor, which is itself a left-adjoint.* □

$$\begin{tikzcd}
 & \underline{K} \arrow[drr, "\underline{r}"] & & \\
K \arrow[ur, "k"] \arrow[drr, "r"'] & & & \underline{N} \\
 & \underline{H} \arrow[uu, "\underline{t}"'] \arrow[drr, "id"] & N \arrow[ur, "n"'] & \\
H \arrow[uu, "t"] \arrow[ur, "h"'] \arrow[drr, "r'"'] & & & \underline{J} = \underline{H} \arrow[uu, "\underline{r} \circ \underline{t}"'] \\
 & & J \arrow[uu, "t_2"] \arrow[ur, "j"] &
\end{tikzcd}$$

Fig. 6. Second Migration Step

[15] Recall that J is the domain of j in $\mathcal{C}$.

[16] For a graph J we write $x \in J$ to depict a vertex or an edge x of graph J.

We sketch how to construct the involved functors in Corollary 1 explicitly: In the first migration step the pullback of two arrows $t_1 : i \to m$ and $l : k \to m$ in $\mathcal{C}^2$ is constructed "pointwise". Assume that this yields an instance graph (with component structure) h typed by the pair $(t, \underline{t})$ in the model graph k. To complete this situation with the additionally depicted arrows in Figure 6, one defines the component structure of the resulting instance graph J to be equal to the one of H, i.e. $\underline{J} := \underline{H}$. The main idea is to prevent identification of equally typed instance elements in H by defining J as follows: For the family $(N_y := r(t(h^{-1}(\{y\}))))_{y \in \underline{H}}$ of sets, define $J := \uplus_{y \in \underline{H}} N_y$. Source and target relations as well as identity edges in J can be defined in a natural way. Furthermore, for each $y \in \underline{H}$, let $r'_y : h^{-1}(\{y\}) \to N_y$ be the restriction of $r \circ t$ and define $r' : H \to J$ by $r'(x) := r'_y(x)$ whenever $h(x) = y$. It is easy to see that r' is a surjective homomorphism with

$$ker(r') \subseteq ker(h) \cap ker(r \circ t)$$

yielding unique arrows j and t_2 such that (by the fundamental homomorphism theorem) the bottom and front squares commute in Figure 6. Finally, an easy calculation yields $\underline{r} \circ \underline{t} \circ j \circ r' = n \circ t_2 \circ r'$, which shows that the right square commutes. Furthermore, in [13] we proved that (r', id) is the unit of the adjunction involving the inclusion functor $\mathscr{I} : \mathcal{U} \to \mathcal{C}^2 \downarrow n$. Thus, $(t_2, \underline{r} \circ \underline{t})$ is the result of the correct migration functor. It can now easily be calculated that the migration in the example in Section 2 reflects this construction: Components are solid frames, J in Figure 4 consists of three disjoint sets ($\{:A,:B,:a\}$, $\{:c\}$, and $\{:C\}$), and t_2 is the migration result.

4.2 Data Models Axiomatically

In this section, we want to sketch a different example in order to demonstrate the use of the framework in other settings. Tight associations of section 4.1 are replaced by containment relations on the one hand and associations with target cardinality 0..1 on the other hand. It can be shown that the effects of applying our construction in this setting yield slightly different results. This may enable tools to migrate data due to different merging modes. Examples can be found in [24].

Let $\mathcal{C} = SET^{\Sigma}$ be as before and assume that graphs $G = (G_V, G_E, src_G, tgt_G)$ implement a distinguished *containment* relation $c \subseteq G_V \times G_V$. Let $\mathcal{C}_c$ be the extended category together with morphisms which respect containments. For a typed instance graph $I \xrightarrow{type} N \in \mathcal{C}_c \downarrow N$, there are the following axiomatic restrictions on the multiplicity and the aggregation level of associations:

$$\forall e_1, e_2 \in I_E : (src_I(e_1) = src_I(e_2) \land type(e_1) = type(e_2)) \Rightarrow e_1 = e_2 \quad (3)$$

expresses restricted target cardinality and the axiom

$$\forall v_1, v_2 \in I_V : (c(v_1, v_2) \land type(v_1) = type(v_2)) \Rightarrow v_1 = v_2 \quad (4)$$

claims that an object contains no other statically equal typed object[17]. This gives rise to the following definition of the subcategory $\mathcal{U}_N$ of valid typings:

$$\mathcal{U}_N := \{\ I \xrightarrow{type} N \ \mid\ (3) \text{ and } (4) \text{ are valid in } I\}$$

[24] proves that $\mathcal{U}_N$ meets the prerequisites of Theorem 2. Thus we obtain.

Corollary 2. *Let* $\mathfrak{TR} = \ M \xleftarrow{l} K \xrightarrow{r} N$ *be a transformation rule in* $\mathcal{C}_c$ *and* $\mathcal{U}_N$ *be the subcategory of valid typings of* N*. Then there exists a correct, essentially unique* $\mathfrak{TR}$*-induced migration functor.* $\square$

5 Related Work

The idea to adapt models without invalidating existing instance landscapes is not new. Our approach differs from schema evolution methods [11,15,21], because we underpin our work with a categorical framework and invent a formal correctness criterion.

In Part IV of [6] exogenous model transformations induced by cospan correspondence between meta models are performed. See also [27] for a comparative study. Triple graph grammars [14] is an approach for co-evolution based on graph transfomations using joined meta-models. [12,23] present similar formal approaches for model transformation which induce instance migrations based on category theory. In some applications, the question arises how to guarantee that the application of rules preserves certain subcategories [2].

The importance of pullback constructions that induce model transformations from meta-model correspondences are stressed in [4]: Pullbacks of typed instances (models in this case) along a map from target meta-model to some derived artifact (having the source meta-model embedded) yield the migration result. Again, the governing rule is a cospan and no span as in our approach.

[22] compares model migration tools for model adjustments due to evolving meta-models. User-oriented migration strategies for automated correct migrations are defined and the quality of the tools is determined. Correctness is based on user requirements formulated in terms of strategies.

6 Conclusion and Outlook

With the framework presented above, we showed that automatic instance migration induced by model transformations is a functor $\mathscr{M}$ between categories of typed instances. Moreover, we stated sufficient conditions for correctness and uniqueness (Theorem 2). There are at least three directions for future research:

Model changes in more general categories: Model changes also occur in the context of typed attributed graph transformations. Moreover, semantics of pred-

[17] Containment of objects is allowed, if the same type may dynamically be assigned.

icates in generalized sketches are defined via typed instances. It is a goal to transfer the present results to these more general cases.

Combination of migrations: Composition of migration functors describes sequential application of several migration steps. It is important to investigate, under which conditions these steps are *sequentially independent* [6]. Furthermore, an analysis of *parallel independence* provides conditions to indicate situations in which the steps are mutually exclusive, such that more modellers can work simultaneously. In addition to that, it is a goal to support rule definition by visual languages and rule execution by transformation environments.

Migration of software: If models change, not only data has to be migrated, but operations and methods have to be adjusted, too. If operation effects are specified by graph rewriting rules, their induced transaction is a double-pushout diagram d in the category $\mathcal{C}\downarrow M$ of typed graphs. Then a transformation rule $\mathfrak{TR}$ induces an automatic transformation of d to a double-pushout-diagram d' in the category $\mathcal{C}\downarrow N$, because the left-adjoint $\mathscr{M}$ (by Theorem 1) preserves co-limits. It is a goal to formulate this idea more precisely.

References

1. Adámek, J., Herrlich, H., Strecker, G.E.: Abstract and Concrete Categories: The Joy of Cats. Free Software Foundation (2004)
2. Biermann, E., Ermel, C., Taentzer, G.: Lifting parallel graph transformation concepts to model transformation based on the Eclipse modeling framework. Electronic Communications of the EASST 26 (2010)
3. Corradini, A., Heindel, T.: und Barbara König, F.H.: Sesqui-pushout rewriting. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 30–45. Springer, Heidelberg (2006)
4. Diskin, Z., Dingel, J.: A metamodel independent framework for model transformation: Towards generic model management patterns in reverse engineering. In: Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies for Reverse Engineering (ateM 2006). Johannes-Gutenberg-Universität Mainz (2006)
5. Diskin, Z., Wolter, U.: A diagrammatic logic for object-oriented visual modeling. Electronic Notes in Theoretical Computer Science 203(6), 19–41 (2008)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
7. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Springer, Heidelberg (1985)
8. Fiadeiro, J.L.: Categories for Software Engineering. Springer, Heidelberg (2005)
9. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
10. Freyd, P.: Aspects of topoi. Bulletin of the Australian Mathematical Society 7, 1–76 (1972)
11. Hainaut, J.L., Tonneau, C., Joris, M., Chandelon, M.: Transformation-based database reverse engineering. In: Elmasri, R.A., Kouramajian, V., Thalheim, B. (eds.) ER 1993. LNCS, vol. 823, pp. 364–375. Springer, Heidelberg (1994)

12. Hermann, F., Ehrig, H., Ermel, C.: Transformation of type graphs with inheritance for ensuring security in e-government networks. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 325–339. Springer, Heidelberg (2009)
13. König, H., Löwe, M., Schulz, C.: Functor semantics for refactoring-induced data migration. Tech. Rep. 02007/01, Fachhochschule für die Wirtschaft Hannover (2007)
14. Königs, A., Schürr, A.: Tool integration with triple graph grammars – A survey. Electronic Notes in Theoretical Computer Science 148(1), 113–150 (2006)
15. Lee, S.-W., Ahn, J.-H., Kim, H.-J.: A schema version model for complex objects in object-oriented databases. Journal of Systems Architecture 52(10), 563–577 (2006)
16. Löwe, M.: Graph rewriting in span-categories. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 218–233. Springer, Heidelberg (2010)
17. Löwe, M., König, H., Schulz, C., Peters, M.: Refactoring information systems – Handling partial composition. Electronic Communications of the EASST 3 (2006)
18. McLarty, C.: Elementary Categories, Elementary Toposes. Clarendon Press (1995)
19. Mens, T.: On the use of graph transformations for model refactoring. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 219–257. Springer, Heidelberg (2006)
20. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152, 125–142 (2006)
21. Roddick, J.F.: A survey of schema versioning issues for database systems. Information and Software Technology 37(7), 383–393 (1995)
22. Rose, L.M., Herrmannsdoerfer, M., Williams, J.R., Kolovos, D.S., Garcés, K., Paige, R.F., Polack, F.A.C.: A comparison of model migration tools. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 61–75. Springer, Heidelberg (2010)
23. Rutle, A., Wolter, U., Lamo, Y.: A diagrammatic approach to model transformations. In: Proceedings of the 2008 Euro American Conference on Telematics and Information Systems (EATIS 2008), pp. 1–8. ACM (2008)
24. Schulz, C.: Transformation Objektorientierter Systeme basierend auf algebraischen Graphtransformationen. Ph.D. thesis, Technische Universität Berlin, Berlin, Deutschland (2010)
25. Schulz, C., Löwe, M., König, H.: A categorical framework for the transformation of object-oriented systems: Models and data. Journal of Symbolic Computation 46(3), 316–337 (2011)
26. Taentzer, G., Beyer, M.: Amalgamated graph transformations and their use for specifying AGG – An algebraic graph grammar system. In: Ehrig, H., Schneider, H.-J. (eds.) Dagstuhl Seminar 1993. LNCS, vol. 776, pp. 380–394. Springer, Heidelberg (1994)
27. Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varró, D., Varró-Gyapay, S.: Model transformation by graph transformation: A comparative study. In: Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2005 (2005)

SPARKSkein: A Formal and Fast Reference Implementation of Skein

Roderick Chapman[1], Eric Botcazou[2], and Angela Wallenburg[1]

[1] Altran Praxis Limited, 20 Manvers Street,
Bath BA1 1PX, U.K.
[2] AdaCore, 46 rue d'Amsterdam,
75009 Paris, France
{rod.chapman,angela.wallenburg}@altran-praxis.com,
botcazou@adacore.com

Abstract. This paper describes SPARKSkein – a new reference implementation of the Skein cryptographic hash algorithm, written and verified using the SPARK language and toolset. The new implementation is readable, completely portable to a wide-variety of machines of differing word-sizes and endian-ness, and "formal" in that it is subject to a proof of type safety. This proof also identified a subtle bug in the original reference implementation which persists in the C version of the code. Performance testing has been carried out using three generations of the GCC compiler. With the latest compiler, the SPARK code offers identical performance to the existing C reference implementation. As a further result of this work, we have identified several opportunities to improve both the SPARK tools and GCC.

Keywords: Skein, Hash, SHA-3, SPARK, Theorem Proving, GCC, Optimization, Verification, Security.

1 Introduction

This paper describes SPARKSkein – a new reference implementation of the Skein cryptographic hash algorithm [1], written and verified using the SPARK[1] language and toolset.

This work started out as an informal experiment to see if a hash algorithm like Skein could be realistically implemented in SPARK. The goals of the implementation were as follows:

- **Readability.** We aimed to strike a reasonable balance of readability and performance. The code should be "obviously correct" to anyone familiar with the Skein specification and/or the existing C reference implementation.
- **Portability.** SPARK has a truly unambiguous semantics, making it a very portable language. Therefore, we aimed for a single code-base that was portable and correct on all target machines of any word-size and endian-ness.

[1] The SPARK Programming Language is not sponsored by or affiliated with SPARC International Inc and is not based on the SPARC® architecture.

- **Performance.** We hoped that the performance of the SPARK code would be close to or better than the existing C reference implementation. The conjecture here is that code that is both correct and type-safe can also be fast. If we failed on the performance front, then we hoped to at least understand *why* as a way of promoting further work on compiler optimization for SPARK.
- **Formality.** The SPARK verification tools offer a full-blown implementation of Hoare-Logic style verification, supported by both an automatic and an interactive theorem prover. We aimed to prove at least type-safety (i.e. "no exceptions") on the SPARKSkein code. There seems to be a belief that "formal is slow" in programming languages, thus justifying the continued use of low-level and type-unsafe languages like C in anything that is thought to be in any way real-time or performance critical. This work aims to provide evidence to refute this view.
- **Empirical.** We aimed to make the experiment *empirical* in that all the code, data, and tools are freely available to the scientific community.

Could we do it? Could we produce code that is formal, provable, readable, portable *and* fast?

2 Skein

Skein [1] is one of the candidate algorithms in the third and final round of the competition to design the future standard hash algorithm that will become known as SHA-3 [12]. Skein is designed for cryptographic strength, portability, and performance, although it is particularly designed for efficiency on 64-bit little-endian machines, such as x86_64, which dominate in desktop computing. Skein is fully defined in [13] using an algorithmic specification accompanied by proofs of a number of key security properties.

3 SPARK

This section provides a brief overview of SPARK and its capabilities. SPARK-aware readers may skip ahead.

SPARK is a contractualized subset of Ada. The contracts embody data- and information-flow, plus the classical notions of pre-condition, post-condition and assertions in code. The language is designed to have a wholly unambiguous semantics – there are *no* unspecified or undefined language features in SPARK – meaning that static analysis can be both fast and sound. The contract language is designed for wholly static verification through the generation of Verification Conditions (VCs) and the use of theorem proving tools. SPARK is well-known in the development of safety-critical systems, but is also being used in some high-grade secure applications, where its properties and verification system have proven useful.

As a subset of Ada, SPARK can be compiled by any standard Ada compiler. The contracts look like comments to an Ada compiler, but are an inherent part of the language as far as the verification tools are concerned. The unambiguous semantics also means that a SPARK program has the same meaning regardless of choice of compiler or target machine – endian-ness, word-size, and so on just don't matter at all.

There are four main tools. The *Examiner* is the main static analysis engine – it enforces the language subset and static semantics, and then goes on to perform information-flow analysis [2]. The Examiner includes a Verification Condition Generator (VCG) – essentially an implementation of Hoare's assignment axiom – that produces VCs in a logic suitable for an automated theorem prover called the *Simplifier*[2]. This is an heuristics-driven automated prover. For VCs that the Simplifier can't prove, we have the *Checker* – an interactive proof assistant based on the same core inference engine. Finally, a tool called *POGS* collates and reports the status of each VC for the entire program.

Further details about SPARK can be found in the SPARK textbook [3] and the Tokeneer on-line tutorial [4]. The GPL edition of the SPARK toolset is freely available under the terms of the GPL [5].

4 Implementing SPARKSkein

The current implementation delivers the main Skein hash algorithm with a 512-bit block-size. For the purposes of this exercise, the other block-sizes and uses of Skein were not relevant.

The coding was straightforward. The main challenge was in understanding the Skein specification and the existing C implementation in sufficient detail to produce a correct SPARK implementation.

One challenge arises in laying out the structure of the Skein "Tweak Words" record. In the C implementation, these are just an array of two 64-bit words, but in SPARK we chose to declare this as a record type with named fields for ease of reading. This means that the layout of this record type has to be different on big-endian and little-endian machines. To do this, a representation clause specifies the bit-numbering required, but depends on the constant System.Default_Bit_Order to get the correct order and layout for the target machine.

To illustrate the difference in coding style, consider the initialization of the hash context in Skein_512_Init. In the C reference implementation, this looks like a function call:

```
Skein_Start_New_Type(ctx,CFG_FINAL);
```

Closer inspection, though, reveals that this is actually a pre-processor macro:

```
#define Skein_Start_New_Type(ctxPtr,BLK_TYPE)
{ Skein_Set_T0_T1(ctxPtr,0,SKEIN_T1_FLAG_FIRST |
SKEIN_T1_BLK_TYPE_##BLK_TYPE); (ctxPtr)->h.bCnt=0; }
```

This, in turn, refers to the macro Skein_Set_T0_T1. The whole thing expands out into:

[2] Not to be confused with Greg Nelson's better-known Simplify prover.

```
{ { {(ctx)->h.T[0] = ((0));};
{(ctx)->h.T[1] = (((((u64b_t) 1 ) << ((126) - 64)) |
((((u64b_t) (( 4))) << ((120) - 64)) | (((u64b_t) 1 ) <<
((127) - 64)))));}; };
(ctx)->h.bCnt=0;
};
```

which is actually 3 assignment statements, with the various shifting/masking constants picked to get the correct endian-ness for the target machine.

In SPARK, this code becomes a procedure, which treats the Context as a record object that can be assigned to. The whole thing comes out as two assignment statements:

```
Ctx.Tweak_Words :=
   Tweak_Value'(Byte_Count_LSB => 0,
                Byte_Count_MSB => 0,
                Reserved       => 0,
                Tree_Level     => 0,
                Bit_Pad        => False,
                Field_Type     => Field_Type,
                First_Block    => First_Block,
                Final_Block    => Final_Block);
Ctx.Byte_Count := 0;
```

which we argue is more readable. All the complexity of the endian-ness and the layout of the record are hidden in the representation clause, and the compiler takes care of generating the required shifting and masking instructions to construct the correct value.

SPARK naturally supports nesting of subprograms (as in all Pascal-family languages) so this allows a natural top-down decomposition of the main operations into local procedures. This decomposition aids readability, but has a negligible impact on performance, assuming a compiler is able to inline the local procedures.

As far as possible, the implementation follows the structure of the reference C implementation, so anyone familiar with that version should be able to read and follow the SPARK code.

We also added a package Skein.Trace that produces debugging output in *exactly* the same format as the functions in the C code's skein_debug.c, so automatic comparison of debug output would be possible. This proved very useful in testing the output of the SPARK version side-by-side with the C.

5 Verification of SPARKSkein

We have verified SPARKSkein in various ways: using the SPARK static verification tools, testing using the published reference test vectors, structural coverage analysis, testing for portability on as many differing machines that we could lay our hands on, and performance testing. These sections summarize the results of these activities.

5.1 Static Verification and Proof

The SPARKSkein code passes all the analyses and verification implemented by the Examiner with no errors. Additionally, we generated VCs for *type-safety*. This means we prove that a program could never raise an exception at run-time through the failure of a type-safety check, such as a buffer overflow, division-by-zero, numeric overflow and so on. The proof of type-safety essentially proves that a program remains in a well-defined state and would never raise exceptions for any possible input data that meets the stated top-level pre-conditions. A benefit of type-safety proof is that it can detect subtle corner cases such as this.

The implementation produces 367 verification conditions, of which 344 (93.7%) are proven automatically by the Examiner or Simplifier. Of these 344, 6 require the insertion of user-defined lemmas into the theorem-prover. Such user-defined lemmas must be subject to careful review, or validation with the Checker, as they have the potential to introduce unsoundness into the proof. The remaining 23 verification conditions are proved using the Checker, requiring some human assistance.

Prover Says No – a Bug is Discovered

The subprogram Skein_512_Final caused some problems, and led to the discovery of a subtle corner-case bug.

The finalization algorithm uses the number of bits of hash requested to compute how many bytes of hash are required, and therefore how many blocks of data are needed. A loop then iterates to generate the required number of blocks. This loop has to iterate at least once, or else no output would result. This requirement was expressed as a type-invariant in SPARK in that the number of output blocks has to be at least one.

The offending fragment of code is:

```
Byte_Count := (Local_Ctx.H.Hash_Bit_Len + 7) / 8;
```

where the "+" operator is modulo 2^{64}.

The need to have at least one block comes out as a VC with conclusion:

$$((\texttt{Local\_Ctx.H.Hash\_Bit\_Len} + 7) \bmod 2^{64}) / 8 > 0$$

which the theorem prover refused to prove for our first implementation – most obviously because it's not true!

The problem is that if the requested Hash_Bit_Len set by Skein_512_Init is sufficiently large (i.e. near 2^{64}), then the "+ 7" overflows to be near zero which, when divided by 8, *is* zero.

This bug is unlikely to happen in reality, based on the assumption that no-one would ask for a hash nearly 2^{64} bits long, but it does illustrate the theorem-prover's ability to sniff out such subtle corner cases that typically elude testing, review or other forms of verification.

The correction is simple enough – we simply limit the range of acceptable hash bit lengths to a maximum of $2^{64} - 8$, so the overflow is avoided. This is encoded in SPARK as a subtype called Hash_Bit_Length, declared in the package specification and then used as the parameter for Skein_512_Init.

In the C reference implementation, this bug persists and the code produces no blocks of output (returning a pointer to an undefined block of memory) for this case.

Reflections on the Proof

The 344 automatically discharged proofs were harder (and slower) than expected. This owes to the prevalence of "modulo N" arithmetic in the VCs. Crypto algorithms tend to do most things using "unsigned" or (in SPARK terminology) "modular" types, which exhibit modular operators like "+" that wrap-round. In the world of proof, this generates VCs that have "mod N" appended to the end of nearly every expression. We also chose to index array types with modular integers, so even innocuous operations like incrementing an array index variable resulted in a "mod N" appearing in the VC. Theorem-provers are notoriously poor with such things - ours included – so the 93.7% of VCs proved automatically is acceptable but offers some room for improvement.

The 23 VCs that remained undischarged by the Simplifier were difficult to prove in the Proof Checker. The main problem is finding a sufficiently strong pre-condition or loop-invariant for the offending code. These tend to have a Goldilocks-like tendency – they mustn't be too strong, mustn't be too weak, but "just right." When the Simplifier fails to prove a VC, it is not always clear why. Perhaps there really is a bug in the code, and the VC has a counter-example? Perhaps the VC really is true, but the Simplifier is just not clever enough to find the proof? Perhaps the VC is unprovable because a pre-condition or invariant is too weak? This last case is particularly annoying – a long session with the Checker can result in merely finding that a VC isn't provable at all. It then isn't always clear exactly what change to the invariant might help. These weaknesses will be considered further in section 6.

A taste of SMT

In parallel with this work, Jackson [6] has produced ViCToR – a tool that translates SPARK VCs into SMTLib so that they can be processed by contemporary SMT-based solvers. Encouragingly, we have found that both Z3 [7] and Alt-Ergo [8] are capable of proving *all* of the VCs arising from SPARKSkein automatically, with Z3 offering by far the better runtime performance at present. We are currently working on integrating the SPARK tools with ViCToR to offer the option of using the Simplifier, one or more SMT-based prover(s), or some combination of both.

5.2 Reference Test Vectors

The test case in the main program "spec_tests" runs the 3 reference test cases given in version 1.2 of the Skein specification.

The first attempt to run this test case failed – the resulting hashes were wrong, illustrating that type-safe code is not necessarily correct. This problem was traced to a simple typing error in the value of the shifting constant R_512_6_3 which had the incorrect value 34 instead of 43.

With that correction in place, the results were as in the Skein specification. No further defects were discovered.

5.3 Platform Testing

The “spec_tests” program has also been added to AdaCore’s regression test suite for GCC. This suite is run nightly on all architectures (both big-endian and little-endian) and operating systems supported by AdaCore.

Target architectures and operating systems include 32-bit x86 (Windows, Linux, FreeBSD, and Solaris), x86_64 (Windows, Linux, Darwin), SPARC (32- and 64-bit Solaris), HP-PA (HP Unix), MIPS (Irix), IA64 (HP Unix, Linux), PowerPC (AIX), and Alpha (Tru64).

The test passes on all platforms.

5.4 Coverage Analysis

The main program “covertest” is designed to exercise boundary values and structural coverage of the hash algorithm. In particular, these test cases are designed to exercise the Skein_512_Update code with various combinations of data blocks of length less than 1 block, exactly 1 block, between 1 and 2 blocks, exactly 2 blocks and more than 2 blocks. This case also tests various sequences of these blocks to cover the cases where a short block results in data being “left over” in the hash context buffer.

This program can be compiled with GCC’s coverage analysis options switched on, and analysed with gcov. The project file “covertest.gpr” builds the program with these options enabled. A single run of “covertest”, followed by “gcov skein.adb” shows 99.7% statement coverage, with a single warning for exactly 1 uncovered line of code. This line is a type declaration which has no object code associated with it, so this must be a false-alarm from gcov.

5.5 Performance Testing

Achieving acceptable performance, but without sacrificing readability and portability, was a major goal of this experiment. This section reports our findings, comparing the performance of the SPARK code against the existing reference implementation in C.

There is a view that anything “formal” must be “slow.” Languages like Ada with their run-time type checking are often criticized for being “slower than C” and therefore not appropriate for time-critical code such as this. Is this really true?

One conjecture we sought to investigate is that type-safe SPARK code should also be fast. SPARK has several properties that make it suitable for hard real-time programming. Furthermore, SPARK code *should* be amenable to more aggressive optimization than other imperative languages. In particular, in SPARK:

- Functions are always *pure* – they have no side-effects.
- There is absolutely no *aliasing* via pointers or names of variables.
- If type-safety has been proven statically (as in this case), then we can safely compile with all runtime checking disabled and more optimistic assumptions about data validity – hopefully making the generated code smaller, faster and simpler.

These properties *should* be taken advantage of by a compiler – where, for example, an optimization pass could make more optimistic assumptions about SPARK code than it could for C. Is this really true? Can a current version of GCC actually find and exploit these properties of SPARK?

Method

These tests were run on a standard PC with an Intel core i7 860 processor running at 2.8GHz. The machine was running 64-bit GNU/Linux (Debian 5.0.5). We chose a 64-bit OS (and compiler) since Skein is designed to perform well on such machines.

The test case "perftest" was written to mimic the testing strategy and performance measurement approach of the "skein_test" program that is supplied with the reference implementation. In this way, we hoped to get results that were reasonably comparable for the C and SPARK implementations.

We also chose to compile the C and the SPARK with the *same* compiler, in this case GNAT Pro 6.3.2 – a stable derivative of GCC 4.3.5 that compiles both SPARK and C through the same back-end. We compiled the C code at various levels of optimization and took the results for Skein_512 hashing a block of 32768 bytes as our base-line for comparison.

When compiling SPARK, GCC offers some additional options that we can take advantage of, so we exercised these to see the effect. In particular, we used the following Ada-specific options:

-gnato – this compiles with *all* the run-time type checking required by the Ada LRM, including checks for arithmetic overflow. This typically generates the slowest code, so was useful as a base-line for the SPARK code.
-gnatp – this option suppresses *all* run-time type checks in the generated code. This is reasonable for us, since we had, of course, already proved that the code was type-safe – effectively showing that run-time checks could never fail. This gives a run-time and code-generation model close to that of C, so we expected comparable performance of the SPARK and the C with –gnatp at the same level of optimization.
-gnatn – enables inlining of subprograms in the back-end of the compiler.

Results were measured in clocks (measured by the x86's rdtsc instructions) per byte hashed, as per the reference skein_test program. Lower numbers indicate better performance:

GNAT Pro 6.3.2 (GCC 4.3.5)		
Options	**SPARK**	**C**
-O0 -gnato	213.9	N/A
-O0 -gnatp	207.9	172.3
-O1 -gnatp	27.6	37.7
-O1 -gnatp -gnatn	26.8	37.7
-O2 -gnatp -gnatn	25.5	24.7
-O3 -gnatp -gnatn	20.4	20.1

Going further – GNAT Pro 6.4.0w
We then re-ran the experiment with a GNAT Pro wavefront (6.4.0w – a derivative of GCC 4.5.0 built on the 28th July 2010). GCC 4.5.0 includes significant improvements across all phases of the back-end, so we expected to see improvement for both C and SPARK. The results were:

GNAT Pro 6.4.0w		
Options	**SPARK**	**C**
-O0 -gnato	71.1	N/A
-O0 -gnatp	69.9	96.5
-O1 -gnatp	22.2	37.0
-O1 -gnatp -gnatn	20.7	37.0
-O2 -gnatp -gnatn	20.2	19.7
-O3 -gnatp -gnatn	13.4	12.3

Analysis – GNAT Pro 6.3.2 vs GNAT Pro 6.4.0w
Coming from compilers based on back-ends separated by two complete cycles of GCC development (roughly two years), these results are significantly different. It is, however, possible to identify a few common patterns.

First of all, the results are uniformly better with the newer compiler and, at -O1 or above, come from improved alias analysis and dead store elimination. The -O0 level is peculiar: for years, the GCC back-end had been known for its totally unoptimized code generation at this level; this was changed in GCC 4.5 and the effect is clearly visible here. This also explains why SPARK gained so much at -O0: being a more expressive language than C, its raw intermediate representation is more verbose and used to be replicated almost verbatim in the generated code at -O0, thus masking the actual merits of the code. To eliminate this old effect, we'll exclude the results at -O0 in the following comparison of SPARK and C.

SPARK is far ahead at -O1 because, even at these low optimization levels, the Ada compiler generates single-instruction inline code for bitwise operators (e.g. shifts and rotates) on scalars. Being rather conservative at these levels, the standard optimization heuristics prevent such operators from being inlined in C.

The next optimization level, -O2, essentially bridges the inlining gap, with C nudging slightly ahead of SPARK with both compilers.

Level –O3 introduces automatic loop unrolling. This is responsible for the big boost in both languages at -03. This leads to roughly equivalent performances with 6.3.2, but not quite so with 6.4.0w because other effects are exposed with the newer compiler. Specifically, it appears that the improved partial redundancy elimination in loops is more efficient on the C code. The SPARK code also suffers from slightly inferior scalarization of composite types and from too limited store/copy propagation.

Finally, it's worth noting that the big boost at -O3 can be partially retrofitted at lower optimization levels in both languages by manually unrolling the single loop in the procedure Inject_Key in the SPARK code. This loop includes an expensive “mod 9” operator, causing a pipeline stall when enclosed in a loop. Unrolling this loop “by hand” in the source code improves the performance of the SPARK code from 20.2 to 13.3 clocks per byte using GNAT 6.4.0w at –O2, for example.

Improving GNAT Pro 6.4.1

As a result of this analysis, we designed a number of improvements to the Ada middle-end, which translates the Ada front-end’s intermediate language into that expected by the GCC back-end. In particular, these improvements generate GCC intermediate-language that is more amenable to the optimization of partial redundancies, scalarization of composite objects, and store/copy propagation.

These improvements are now included in the GNAT Pro 6.4.1 release of March 2011. With this new compiler, the results look like this:

GNAT Pro 6.4.1, March 2011		
Options	**SPARK**	**C**
-O0 -gnato	70.6	N/A
-O0 -gnatp	69.7	96.4
-O1 -gnatp	22.2	37.0
-O1 -gnatp -gnatn	20.5	37.0
-O2 -gnatp -gnatn	20.0	19.7
-O3 -gnatp -gnatn	12.3	12.3

Analysis – GNAT Pro 6.4.1

The results are essentially identical to those obtained with the 28th July 2010 wavefront, except for the -03 level where SPARK is now on par with C. The original analysis still holds. In particular, the effects of inlining and loop unrolling still dominate here.

SPARK still lags a little behind C at -O2. This comes from a couple of missed Partial Redundancy Elimination opportunities for the SPARK version, which can be traced to an application of the “Unchecked_Conversion” function on a composite object. The application of this function forces the Ada compiler to make worst-case assumptions about the resulting value, preventing a couple of subsequent optimizations from taking place. From this, we derive a simple coding rule for high-performance Ada and SPARK code: don’t use Unchecked_Conversion on composite objects.

The changes made to the compiler are not specifically tuned to the Skein code or any other particular benchmarks, so they should benefit Ada programs in general.

6 Further Work and Challenges

This section presents a few ideas that might warrant further work or form challenges for other tools and research groups.

6.1 GCC

For the GNAT compiler, the results conform to the general trend observed over the years: the aggressive optimizations implemented in the GCC back-end are initially tuned to the C family of languages. A little more work is required in order to make them as effective in SPARK or Ada, and the end result is almost always generated code equally well optimized whatever the source language. Our improvements to GCC resulting from this project will appear in a future release of GCC from the FSF.

6.2 The SPARK Tools

For the SPARK tools, several improvements have been identified as a result of this work. Most notably, the procedure Skein_512_Update causes extremely poor performance from the Simplifier – taking nearly an hour to simplify on the Core i7 machine used for testing. We hope to identify and correct this matter in a future release of the SPARK Tools.

The proofs that require interaction with the Checker provide a rich source of examples for further improvement of the Simplifier's proof tactics, particularly in the area of modular arithmetic.

As we noted above, the potential to exploit SMT-based solvers offers a notable improvement in both performance and completeness of proof. We are currently benchmarking these provers on substantially larger programs than SPARKSkein to determine which prover (or combination of provers) offers the most benefit.

Finally, to improve the insight and feedback arising from failed proof attempts, we are actively pursuing research on counter-example finding, initially focusing on the use of answer-set programming [9] supported by SMT-based provers.

6.3 Comparison with Other Verification Tools

The Skein code could be used as a "Challenge Problem" for other verification tools. Given the SPARK proofs of the code, it could be used as a test-case to measure the performance and true false-alarm rate of other tools. In particular, we would like to assess the ability of other tools to rediscover the pre-conditions and invariants that proved troublesome to find by hand. Similarly, tools such as VeriFast [10] or Microsoft's VCC [11] could be used to recreate a proof of type safety for the C reference implementation.

7 Conclusions

Returning to our original goals, it seems the project can be judged a success. An algorithm like Skein can be written in a "formal" language like SPARK without sacrificing readability and performance. Portability can only be judged a success – a single set of sources with no macros, “ifdefs” or pre-processing gives identical results on every architecture and OS we could find. SPARK’s type-safe nature allowed us to “turn up the dials” on the compiler’s optimizers with confidence. The SPARK code did perform better than C at –O1, reflecting more aggressive inlining in the Ada front-end at that level. At –O2 and –O3, C pulls ahead by a small margin using compilers that were available in the middle of Summer 2010. As a result, we identified and implemented improvements in more recent GCC releases that show the SPARK code having essentially identical performance to the C.

Acknowledgements. The authors would like to thank Doug Whiting and Jesse Walker of the Skein design team. Doug patiently answered questions about the C implementation. Jesse offered valuable comments on an early draft of this paper.

Obtaining SPARKSkein. The SPARKSkein sources, test cases, and proofs are available from the Skein website [1]. The GPL editions of the SPARK Toolset and GNAT compilers are available from [5].

References

1. Skein project homepage, `http://www.skein-hash.info/`
2. Carré, B., Bergeretti, F.: Data- and Information-Flow Analysis of While Programs. ACM Transactions on Programming Languages and Systems 7(1), 36–61 (1985)
3. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley (2003) (reprinted in 2007) ISBN 978-0-321-13616-0
4. Tokeneer Discovery: A SPARK Tutorial, `http://www.adacore.com/home/products/sparkpro/tokeneer/discovery/`
5. SPARK GPL Edition site, `http://libre.adacore.com/`
6. Jackson, P.B., Ellis, B.J., Sharp, K.: Using SMT Solvers to Verify High-Integrity Programs. In: 2nd International Workshop on Automated Formal Methods, AFM 2007, Atlanta, Georgia, USA (2007), `http://homepages.inf.ed.ac.uk/pbj/`
7. Z3: An efficient theorem prover. Microsoft Research, `http://research.microsoft.com/enus/um/redmond/projects/z3/`
8. Alt-Ergo website, `http://alt-ergo.lri.fr/`
9. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
10. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (August 2008)
11. VCC: A Verifier for Concurrent C, `http://research.microsoft.com/en-us/projects/vcc/`
12. National Institute of Standards and Technology, Computer Security and Resource Center, Cryptographic Hash Algorithm Competition, `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`
13. The Skein Hash Function Family, Ferguson, N., et al., `http://www.skein-hash.info/sites/default/files/skein1.1.pdf`

Full Abstraction at Package Boundaries of Object-Oriented Languages*

Yannick Welsch and Arnd Poetzsch-Heffter

University of Kaiserslautern, Germany
{welsch,poetzsch}@cs.uni-kl.de

Abstract. We develop a fully abstract trace-based semantics for sets of classes in object-oriented languages, in particular for Java-like sealed packages. Our approach enhances a standard operational semantics such that the change of control between the package and the client context is made explicit in terms of interaction labels. By using traces over these labels, we abstract from the data representation in the heap, support class hiding, and provide fully abstract package denotations. The soundness and completeness of our approach is proven using innovative simulation techniques.

1 Introduction

Systems, components, and libraries have to evolve over time to meet new requirements. In an object-oriented setting, such evolution steps affect the classes used in the implementation. An important aspect for safe evolution is the ability to modularly check for *compatibility*, i.e., whether a new version has the same behavior as the old one in *all* program contexts in which the old version can be used. In particular, every refactoring should guarantee compatibility. Mutual compatibility corresponds to the classical notion of *(contextual) equivalence*: Two sets of classes are equivalent if they exhibit the same operational behavior in every possible context. Proving compatibility or equivalence is challenging because

(1) the number of possible contexts is infinite and contexts are complex
(2) the states and heaps can be significantly different between the versions.

To meet these challenges, we exploit denotational methods. A denotational semantics for classes is called *fully abstract* [11,13] if classes that have the same denotation are exactly those that are contextually equivalent. In particular, a fully abstract semantics has to abstract from states and heaps to meet challenge (2) above. Proving that two sets of classes are equivalent in the (fully abstract) denotational setting amounts to proving that they have the same denotation.

The central contribution of this paper is the design of such a fully abstract semantics for packages of a Java subset. Furthermore, the paper provides a detailed explanation of the full abstraction proof. To explain our approach in more detail, we consider the following two versions of package xutil:

* This research is funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models.

```
package xutil; import java.util.∗;



public class Bag implements Collection {
  private ArrayList mList;
  public boolean addAll(Collection c){ BODY }
  ...
}
```

```
package xutil; import java.util.∗;
public interface IBag extends Collection {
  public boolean addAll(Collection c);
  ... }
public class Bag implements IBag {
  private MyList mList;
  public boolean addAll(Collection c){ BODY2 }
  ... }
class MyList { IMPL }
```

The goal is to show that the version on the right-hand side has the same behavior as the one on the left-hand side in all possible program contexts. In particular, we address the following language-related challenges:

– change of implementation, e.g., BODY2 instead of BODY, MyList instead of ArrayList
– change of subtype hierarchy, e.g., interface IBag is added
– non-public, encapsulated classes, e.g., the new class MyList is package local
– use of imported types in signatures (e.g., interface Collection in signature of method addAll) and in the implementation (e.g., ArrayList is imported)
– inheritance and casts (not illustrated).

Approach. In our approach, the denotation of a package (or set of classes and interfaces) is expressed by the interactions between code belonging to the package and code belonging to the context. It is defined in *two* steps starting from a standard operational semantics. In the first step, the operational semantics is augmented in a way that the interactions can be made explicit. In the second step, traces of interaction labels are used to semantically characterize the package behavior. A non-trivial aspect is the treatment of inheritance, because with inheritance, some code parts of a class/object might belong to the context and other parts to the package under investigation.

Using traces allows abstracting from the state and heap representation in the old and new version. It solves challenge (2). To obtain a finite representation of all contexts and solve challenge (1), we construct a nondeterministic *most general context* that exactly exhibits the possible behavior of contexts. Using an operational semantics as a starting point has the advantage that we can use simulation relations applied to standard configurations (i.e., heap, stack) for the full abstraction proof. Furthermore, it provides a direct formal relation to Hoare-like program logics and standard techniques for static program analysis.

Related Work. Banerjee and Naumann [4] presented a method to reason about whole-program equivalence in a Java subset. Under a notion of confinement for class tables, they prove equivalence between different implementations of a class by relating their (classical, fixpoint-based) denotations by simulations. Silva, Naumann, and Sampaio [15] extend this work to prove several refactoring laws for whole hierarchies of classes.

Similar to our work, Jeffrey and Rathke [8] give a fully abstract trace semantics for a Java subset with a package-like construct. However, they do not consider inheritance, down-casting and cross-border instantiation. Using similar techniques, Ábrahám et al. [2] give a fully abstract semantics for a concurrent class-based language (without inheritance and subtyping).

Whereas we use simulation techniques to prove full abstraction, other authors apply (bi-)simulations to relate two program parts. This technique was first used by Hennessy and Milner [7] to reason about concurrent programs. Sumii and Pierce used bisimulations which are sound and complete with respect to contextual equivalence in a language with dynamic sealing [16] and in a language with type abstraction and recursion [17]. Koutavas and Wand, building on their earlier work [9] and the work of Sumii and Pierce, used bisimulations to reason about the equivalence of *single* classes [10] in different Java subsets. The subset they considered includes inheritance and down-casting. Their language, however, neither considers interfaces nor accessibility of types.

Outline. In the following, we illustrate our method to develop fully abstract denotational semantics for Java-like packages. The remainder of this paper is structured as follows. Section 2 introduces the formalized language LPJava and the notion of source compatibility. Section 3 gives the operational and trace semantics of LPJava and presents the full abstraction result. Section 4 shows how full abstraction can be proved by simulation relations and Section 5 gives a general outline on how to prove two components compatible. Section 6 presents directions for future work and concludes. For proofs and further details on the formalization, we refer to the extended report [18].

Notations. We use the *overbar* notation $\overline{x}$ to denote a finite list and the *hat* notation $\widehat{x}$ to denote a set. The empty list and set are denoted by $\bullet$ and the concatenation of list $\overline{x}$ and $\overline{y}$ is denoted by $\overline{x} \cdot \overline{y}$. Concatenation is sometimes implicit by writing terms in juxtaposition. Single elements are implicitly treated as lists/sets when needed. The function last$(\dots)$ returns the last element of a list. The expression $\mathcal{M}[x \mapsto y]$ yields the map $\mathcal{M}$ where the entry with key x is updated with the value y, or, if no such key exists, the entry is added. The empty map is denoted by $\varnothing$ and dom$(\mathcal{M})$ and rng$(\mathcal{M})$ denote the domain and range of the map $\mathcal{M}$.

2 Formalization of LPJava and Source Compatibility

The language considered in the following is a sequential object-oriented language called LPJava (see Fig. 1). It has interfaces, classes and subtyping. To consider the additional challenge of inheritance and type hiding, it also has subclassing and a package system. Classes can extend other classes, and can be declared either package-local or public. Primitive data types (like **int**, etc.) are not considered as they do not provide additional insight. For simplicity, all methods are assumed to be public and all fields to be private. LPJava also allows explicit casting, which leads to more distinguishing power from class contexts. The operator $(p.t)E_1 : E_2$ encodes both an *instanceof* and *cast* operator, i.e., it yields the value of E_2 if the value of E_1 cannot be cast to $p.t$.

We assume that every class has a default constructor. Similar as for Java [5], the default constructor has the same access modifier as its class. A *package* (denoted by Q or R) has a name and consists of a sequence of type declarations. We assume that

$$
\begin{array}{rcl}
K,X,Y & ::= & \overline{Q} \\
Q,R & ::= & \textbf{package}\ p\ ;\ \overline{D} \\
D & ::= & [\textbf{public}]\ \textbf{class}\ c\ \textbf{extds}\ p.c\ \textbf{impls}\ \overline{p.i}\ \{\ \overline{F}\ \overline{M}\ \} \\
 & | & [\textbf{public}]\ \textbf{interface}\ i\ \textbf{extds}\ \overline{p.i}\ \{\ \overline{M}\ \} \\
F & ::= & \textbf{private}\ p.t\ f\ ; \\
M & ::= & \textbf{public}\ p.t\ m(\overline{p.t\ v})\ \big(\ ;\ \mid\ \{\ E\ \}\ \big) \\
E & ::= & x \mid \mathsf{null} \mid \mathsf{new}\ p.c() \mid (p.t)E : E \mid E.f \mid E.f = E \\
 & | & \mathsf{let}\ p.t\ x = E\ \mathsf{in}\ E \mid E.m(\overline{E}) \mid E == E\ ?\ E : E
\end{array}
$$

$$
\begin{array}{rcl}
t & ::= & c \mid i \\
c & \in & \text{class names} \\
i & \in & \text{interface names} \\
p,q & \in & \text{package names} \\
f & \in & \text{field names} \\
m & \in & \text{method names} \\
x & \in & \text{variable names}
\end{array}
$$

Fig. 1. Abstract syntax of LPJava

packages are sealed (cf. [6], Sect. 2), meaning that once a package is defined no new class and interface definitions can be added to the package. Types are fully qualified by their package name.

A *codebase*, which consists of a list of packages, is denoted by K, X or Y. If it satisfies all the well-formedness conditions of the language, i.e., well-formed type hierarchy, well-typedness of all expressions, etc., we write $\vdash K$ (or $\vdash X$, $\vdash Y$) and call such a codebase a *component*. Note that components are definition-complete. To join two codebases into a larger codebase, we write them in juxtaposition (i.e., KX). If we join a codebase K and a component X, we often call K a (class) *context* of X.

A prerequisite for two components to have the same behavior is that whenever the first component can be joined with a context into a larger component, then the second one can be joined as well using the same context. This property,[1] focusing solely on typing and not behavioral aspects, is called *source compatibility* [8]:

Definition 1 (Source compatibility). *A component Y is* source compatible *with a component X if for any codebase K: $\vdash KX$ implies $\vdash KY$.*

It is important to notice that this does not allow automatic checking that a component Y is source compatible with X, because the definition quantifies over an infinite set of contexts. However, a set of checkable conditions that are necessary and sufficient for Y to be source compatible with X can be given. We describe them in the following.[2] The package names occurring in X must exactly be those occurring in Y. Every public type defined in X must appear in Y. For public types of X, every method which is part of the type (declared or inherited) in X must also have a method with the same signature (i.e. same parameter and return type) in Y and vice versa. The subtype hierarchy between public types of X must be maintained in Y.

Theorem 1. *A component Y is source compatible with a component X if and only if the checkable conditions between X and Y described above hold.*

[1] Note that the definition is not symmetric. This allows Y to be a more *refined* version of X.

[2] As this paper focuses on behavioral aspects, we do not explain here why these conditions are necessary and sufficient. However, explanations and proofs can be found in [18].

E	$::=$	$\ldots \mid r$	extd. expressions
r	$::=$	$j \mid \mathsf{null}$	reference
$\mathcal{O}$	$::=$	$j \mapsto H$	heap
H	$::=$	$(V, L, p.c, \overline{r})$	heap entry
$\mathcal{F}$	$::=$	$(E \mid \mathcal{E})_L{:}p.t$	typed stack frame
V	$::=$	$\mathsf{internal} \mid \mathsf{exposed}$	exposure flag
L	$::=$	$\mathsf{ctxt} \mid \mathsf{comp}$	origin location
j	$\in$	object identifiers	

Fig. 2. Semantic entities for LPJava

t	$::=$	$\overline{l}$	trace
l	$::=$	$\mu! \mid \mu? \mid \tau$	label
	$\mid$	$\mathsf{error} \mid \mathsf{halt}$	
μ	$::=$	$\mathsf{call}\ o.m(\overline{v})$	call message
	$\mid$	$\mathsf{rtrn}\ v$	return message
o	$::=$	$j{:}T^{\alpha}$	abstracted object
v	$::=$	$o \mid \mathsf{null}$	label value
T^{α}	$::=$	$\langle \widehat{p.c}, \widehat{p.i}, \widehat{m} \rangle$	abstracted type

Fig. 3. Syntax of traces

3 Trace Characterization of Component Behavior

In this section, we characterize the behavior of a component X in terms of its possible interaction traces with program contexts. We first define the interaction traces of X with a specific program context. Then, we introduce nondeterministic expressions that allow for the definition of most general contexts which simulate all possible contexts. Finally, we state the full abstraction result.

3.1 Enhanced Operational Semantics

We enrich a standard semantics in such a way that the interactions between a component X and its program context K become explicit and call it the enhanced semantics. In particular, we define the traces of interactions between X and K. The rules in Fig. 4 describe the (enhanced) small-step operational semantics of LPJava. Auxiliary functions are defined in Fig. 5.

The operational rules for a component X in a context K are based on a labelled small-step reduction judgement of the form $\zeta \overset{l}{\rightsquigarrow} \zeta'$ with configurations $\zeta = KX, \mathcal{O}, \overline{\mathcal{F}}$. The heap $\mathcal{O}$ is a map from object identifiers to heap entries and $\overline{\mathcal{F}}$ is a list of typed stack frames (see Fig. 2). The configurations are augmented with additional information. However, this does not change the standard operational behavior. The flag L ranges over $\{\mathsf{ctxt}, \mathsf{comp}\}$ and indicates whether entities belong to the context K or to the component X. It is used in stack frames to mark if the code (E or $\mathcal{E}$) that is part of this stack frame originates from X or K. It is also used in heap entries to denote whether the object has been created by code of X or K. The flag V is used in heap entries to denote whether an object created in the context has been exposed to the component or vice versa. Objects are always created internally (see **RI-NewObj**) but can over time be exposed when they are passed from (to) the component to (from) the context.

Stack frames are associated with either the component X or the context K. The topmost stack frame contains an expression E and all other stack frames contain an evaluation context $\mathcal{E}$. An evaluation context $\mathcal{E}$ (see [19]) is an expression with a *hole* $[]$ somewhere inside the expression. We write $\mathcal{E}[E]$ to mean that the hole in $\mathcal{E}$ is replaced by expression E. A hole in $\mathcal{E}$ can only appear at certain positions defined as follows:

$$\begin{aligned}\mathcal{E} ::= & [\,] \mid \mathcal{E}.f \mid \mathcal{E}.f = E \mid r.f = \mathcal{E} \mid \mathsf{let}\ p.t\ x = \mathcal{E}\ \mathsf{in}\ E \mid (p.t)\mathcal{E} : E \mid \mathcal{E}.m(\overline{E}) \\ & \mid r.m(\overline{r}, \mathcal{E}, \overline{E}) \mid \mathcal{E} == E\ ?\ E : E \mid r == \mathcal{E}\ ?\ E : E\end{aligned}$$

We say that X *controls execution* if code of X is executed; otherwise K controls execution. The function currloc(ζ) from Fig. 5 is used to determine who controls execution. An interaction is a change of control (see **RI-Call-Boundary** and **RI-Return-Boundary**). Note that only interactions allocate or deallocate stack frames, i.e., calls within the context or the component are not handled using the stack (see **RI-Call-Intern**). Labels record changes of control. An interaction trace is a finite sequence of labels (Fig. 3). Interaction is considered from the viewpoint of the component. Input labels (marked by ?) express a change of control from the context to the component; output labels (marked by !) express a change from the component to the context. There are input and output labels for method invocation and return, as well as labels for well-formed and abrupt program termination. The labels for method invocation and return include the parameter and result values together with their *abstracted types* (explained later).

A *program context* is a context that has a public class $p.c$ with a main method lang.Object main(), where the class $p.c$ is called a *startup class*. It is executed by calling main. In the following we assume that the startup class is always main.Main and is defined in the context K. The initial configuration $\mathrm{init}\zeta_{KX}$ is then defined as $KX, \mathcal{O}, \mathcal{F} \cdot \bullet$ where $\mathcal{O} \overset{\text{def}}{=} \emptyset[j \mapsto (\mathsf{internal}, \mathsf{ctxt}, \mathsf{main.Main}, \overline{\mathsf{null}})]$ and $\mathcal{F} \overset{\text{def}}{=} E[j/\mathsf{this}]_{\mathsf{ctxt}}$:*lang.Object*, if E is the body of the main method.

Traces. In the following, we consider the traces (i.e. finite lists of labels) which are generated by steps of the operational semantics. To compare traces of different components and with different contexts, we abstract from package local types and types declared in the context. Package local types should not appear in the labels, because different components might use different local types. Types in labels are *abstracted* (see $\mathrm{typeabs}_{KX}(p.c)$ in Fig. 5) to a representation which only preserves the information (1) which public supertypes of $p.c$ belong to the component (2) which of their methods are not overridden by the context. The reason for (1) is that these are the types of X that can be used in cast expressions in the context. Based on the label, it becomes thus clear which cast expressions will succeed and which not. The reason for (2) is that, based on the label, we know the methods that, if invoked with the object as receiver, lead to changes of control. As X defines a finite set of types (denoted by $\mathcal{T}_X$), there are only a finite set of abstracted types that can occur in traces with X. This set is denoted by $\mathcal{T}_X^\alpha$ and can be constructed from X.

To abstract from internal τ steps of a computation, we provide a large step version of the enhanced semantics (denoted $\overset{t}{\Longrightarrow}$) that is defined by:

$$\frac{}{\zeta \overset{\bullet}{\Longrightarrow} \zeta} \qquad \frac{\zeta \overset{t}{\Longrightarrow} \zeta' \qquad \zeta' \overset{\tau}{\rightsquigarrow}_* \zeta'' \qquad \zeta'' \overset{l}{\rightsquigarrow} \zeta''' \qquad l \neq \tau}{\zeta \overset{t \cdot l}{\Longrightarrow} \zeta'''}$$

Every large step represents a finite number of τ steps (denoted by $\overset{\tau}{\rightsquigarrow}_*$, the reflexive, transitive closure of $\overset{\tau}{\rightsquigarrow}$) followed by a non-$\tau$ step. Note that τ does not appear in labels

RI-Internal-Step

$$\frac{\mathcal{O}, E \dashrightarrow^{L}_{KX} \mathcal{O}', E'}{KX, \mathcal{O}, \mathcal{E}[E]_L{:}p.t \cdot \overline{\mathcal{F}} \overset{\tau}{\rightsquigarrow} KX, \mathcal{O}', \mathcal{E}[E']_L{:}p.t \cdot \overline{\mathcal{F}}}$$

RI-NewObj

$$\frac{j \notin dom(\mathcal{O}) \qquad H = (\mathsf{internal}, L, p.c, \overline{\mathsf{null}})}{\mathcal{O}, \mathsf{new}\ p.c() \dashrightarrow^{L}_{KX} \mathcal{O}[j \mapsto H], j}$$

RI-Call-Intern

$$\frac{\mathrm{type}_{\mathcal{O}}(j) = p.c \qquad \langle m, \_, E\rangle \in_{KX} p.c \qquad \mathrm{mdecl}_{KX}(p.c, m) = L}{\mathcal{O}, j.m(\overline{r}) \dashrightarrow^{L}_{KX} \mathcal{O}, E[j/\mathsf{this}, \overline{r}/\overline{v_p}]}$$

RI-Call-Boundary

$$\frac{\begin{array}{c}\mathrm{type}_{\mathcal{O}}(j) = p.c \qquad \langle m, \_, E\rangle \in_{KX} p.c \\ \mathrm{mdecl}_{KX}(p.c, m) = \neg L \qquad l = \mathrm{mcall}_{KX}(j, m, \overline{r}, \mathcal{O}, L) \qquad \mathcal{O}' = \mathrm{expose}(j\overline{r}, \mathcal{O})\end{array}}{KX, \mathcal{O}, \mathcal{E}[j.m(\overline{r})]_L{:}p.t \cdot \overline{\mathcal{F}} \overset{l}{\rightsquigarrow} KX, \mathcal{O}', E[j/\mathsf{this}, \overline{r}/\overline{v_p}]_{\neg L}{:}p'.t' \cdot \mathcal{E}_L{:}p.t \cdot \overline{\mathcal{F}}}$$

RI-Return-Boundary

$$\frac{l = \mathrm{mrtrn}_{KX}(r, \mathcal{O}, L) \qquad \mathcal{O}' = \mathrm{expose}(r, \mathcal{O})}{KX, \mathcal{O}, r_L{:}p'.t' \cdot \mathcal{E}_{\neg L}{:}p.t \cdot \overline{\mathcal{F}} \overset{l}{\rightsquigarrow} KX, \mathcal{O}', \mathcal{E}[r]_{\neg L}{:}p.t \cdot \overline{\mathcal{F}}}$$

RI-Fail

$$\frac{E = \mathsf{null}.f_i \vee E = \mathsf{null}.f_i = r \vee E = \mathsf{null}.m(\overline{v})}{KX, \mathcal{O}, \mathcal{E}[E]_L{:}p.t \cdot \overline{\mathcal{F}} \overset{\mathsf{error}}{\rightsquigarrow} KX, \mathcal{O}, \bullet}$$

RI-Halt

$$\frac{}{KX, \mathcal{O}, r_{\mathsf{ctxt}}{:}p.t \overset{\mathsf{halt}}{\rightsquigarrow} KX, \mathcal{O}, \bullet}$$

Fig. 4. Transition rules for the enhanced small-step semantics, using the helper judgement $\dashrightarrow^{L}_{KX}$ for internal steps that are local to an evaluation context. The rules **RI-Cast**, **RI-Let**, **RI-If**, **RI-FieldSel** and **RI-FieldUp** are not shown but can be found in [18].

of large steps. Large steps always jump to the state right after the next non-τ label has been generated.

Termination ($\zeta\downarrow$) and divergence ($\zeta\uparrow$) are defined in the usual way. We write $\zeta\downarrow$ iff there exists a terminal configuration $\zeta_f = KX, \mathcal{O}, \bullet$ such that $\zeta \overset{t}{\Longrightarrow} \zeta_f$. Note that termination occurs if and only if the last label is either error or halt. We write $\zeta\uparrow$ iff the execution diverges. As can be seen from the transition rules, evaluation is deterministic (up to object naming).

In order to deal with the non-deterministic choice of fresh object identifiers, we introduce (object) renamings. A *renaming* is a bijective relation on object identifiers. We write ρ for such a relation. We can then consider traces equivalent (or related) if they are equal modulo a renaming.

Definition 2 (Related traces). $t_1 \equiv^{\rho} t_2$ *iff the object identifiers appearing at the same positions in the traces are related under* ρ *and the types appearing at the same position are equal. If we are not interested in a particular* ρ*, we omit it for brevity.*

In the following, we use the straightforward generalization of this definition of *equality modulo a renaming* ($\equiv^{\rho}$) to arbitrary syntactic structures. The observable behavior of a program run can then be reduced to the traces it exposes.

$$
\begin{array}{lll}
\mathrm{expose}(r,\mathcal{O}) & \overset{\text{def}}{=} & \begin{cases} \mathcal{O} & \text{if } r = \mathsf{null} \\ \mathcal{O}[j \mapsto (\mathsf{exposed}, L, p.c, \overline{r})] & \text{if } r = j \wedge \mathcal{O}(j) = (\_, L, p.c, \overline{r}) \end{cases} \\
\mathrm{mcall}_{KX}(j,m,\overline{r},\mathcal{O},L) & \overset{\text{def}}{=} & \mathsf{call}\ \mathrm{objectabs}_{KX}(j,\mathcal{O}).m(\mathrm{objectabs}_{KX}(\overline{r},\mathcal{O}))\ \mathrm{fromdir}(L) \\
\mathrm{mrtrn}_{KX}(r,\mathcal{O},L) & \overset{\text{def}}{=} & \mathsf{rtrn}\ \mathrm{objectabs}_{KX}(r,\mathcal{O})\ \mathrm{fromdir}(L) \\
\mathrm{fromdir}(L) & \overset{\text{def}}{=} & \begin{cases} ? & \text{if } L = \mathsf{ctxt} \\ ! & \text{if } L = \mathsf{comp} \end{cases} \\
\mathrm{objectabs}_{KX}(r,\mathcal{O}) & \overset{\text{def}}{=} & \begin{cases} \mathsf{null} & \text{if } r = \mathsf{null} \\ j{:}\mathrm{typeabs}_{KX}(\mathrm{type}_{\mathcal{O}}(r)) & \text{if } r = j \end{cases} \\
\mathrm{typeabs}_{KX}(p.c) & \overset{\text{def}}{=} & \langle \widehat{p.c}, \widehat{p.i}, \widehat{m} \rangle \text{ where } \widehat{p.c} \cup \widehat{p.i} \text{ are the supertypes of } p.c \text{ that are} \\
 & & \text{in pubtypes}(X) \text{ and} \\
 & & \widehat{m} = \{ m \mid m \in \mathrm{methods}_X(\widehat{p.c} \cup \widehat{p.i}) \wedge \mathrm{mdecl}_{KX}(p.c,m) = \mathsf{comp} \} \\
\mathrm{pubtypes}(X) & \overset{\text{def}}{=} & \{ p.t \mid p.t \in \mathcal{T}_X \wedge \mathrm{public}_X(p.t) \} \quad \text{(public types defined in } X\text{)} \\
\mathrm{currloc}(KX,\mathcal{O},\overline{\mathcal{F}}) & \overset{\text{def}}{=} & L \text{ if } \overline{\mathcal{F}} = E_L{:}p.t \cdot \overline{\mathcal{F}'} \quad \text{(location of top of stack)} \\
\mathrm{type}_{\mathcal{O}}(r) & \overset{\text{def}}{=} & \begin{cases} \bot & \text{if } r = \mathsf{null} \\ p.c & \text{if } \mathcal{O}(r) = (\_,\_,p.c,\_) \end{cases} \\
\neg L & \overset{\text{def}}{=} & L' \text{ where } L \neq L' \text{ (similar for } \neg V\text{)} \\
\mathrm{filter}(\mathcal{O},V) & \overset{\text{def}}{=} & \{ j \in \mathcal{O} \mid \mathcal{O}(j) = (V,\_,\_,\_) \} \\
\mathrm{filter}(\mathcal{O},L) & \overset{\text{def}}{=} & \{ j \in \mathcal{O} \mid \mathcal{O}(j) = (\_,L,\_,\_) \} \\
\mathrm{filter}(\mathcal{O},V,L) & \overset{\text{def}}{=} & \mathrm{filter}(\mathcal{O},V) \cap \mathrm{filter}(\mathcal{O},L) \\
\mathrm{visible}(\mathcal{O},L) & \overset{\text{def}}{=} & \mathrm{filter}(\mathcal{O},\mathsf{exposed}) \cup \mathrm{filter}(\mathcal{O},\mathsf{internal},L) \\
\mathrm{abs}_X(\langle \widehat{p.c}, \widehat{p.i}, \widehat{m} \rangle) & \overset{\text{def}}{=} & \langle \widehat{p.c}', \widehat{p.i}', \widehat{m}' \rangle \text{ where } (\widehat{p.c}' \cup \widehat{p.i}') = (\widehat{p.c} \cup \widehat{p.i}) \cap \mathrm{pubtypes}(X) \\
 & & \text{and } \widehat{m}' = \{ m \mid m \in (\widehat{m} \cap \mathrm{methods}_X(\widehat{p.c} \cup \widehat{p.i})) \}
\end{array}
$$

Fig. 5. Helper definitions, where $\mathrm{mdecl}_{KX}(p.c,m)$ yields the location L where the method body has been declared (searching from the class $p.c$ upwards) and $\mathrm{methods}_X(p.t)$ yields the method names of declared and inherited methods in $p.t$.

Definition 3 (Traces). *The* traces *of a component X with a program context K are:* $\mathrm{Traces}(KX) \overset{\mathit{def}}{=} \{ t \mid \exists \zeta : \mathrm{init}\zeta_{KX} \overset{t}{\Longrightarrow} \zeta \}$

Note that $\mathrm{Traces}(KX)$ is closed w.r.t. renaming, i.e., if $t \in \mathrm{Traces}(KX)$ and $t' \equiv t$, then also $t' \in \mathrm{Traces}(KX)$. Furthermore, $\mathrm{Traces}(KX)$ is prefix-closed and only refers to public types in X.

3.2 Most General Context

Although the traces abstract from the context, we still have to consider the traces of all possible program contexts in order to describe the full behavior of a component. This issue is addressed in this section by constructing a most general context κ_X that enables all possible interactions that X can engage in. The context κ_X represents exactly all contexts that X can have. Compared to a concrete context, κ_X abstracts over types, objects, and operational steps. To represent κ_X, we extend LPJava by nondeterministic expressions ($E ::= \ldots \mid \mathsf{nde}$) and corresponding reduction rules (Fig. 6). Nondeterministic

MGC-Skip

$$\frac{}{\mathcal{O}, \mathsf{nde} \dashrightarrow^{\mathsf{ctxt}}_{KX} \mathcal{O}, \mathsf{nde}}$$

MGC-NewObj

$$\frac{j \notin dom(\mathcal{O}) \qquad p.c \in \mathcal{T}_{KX} \qquad \mathrm{public}_{KX}(p.c)}{\mathcal{O}, \mathsf{nde} \dashrightarrow^{\mathsf{ctxt}}_{KX} \mathcal{O}[j \mapsto (\mathsf{internal}, \mathsf{ctxt}, p.c, \overline{\mathsf{null}})], \mathsf{nde}}$$

MGC-PrepareCall

$$\frac{\begin{array}{c} j, r_i \in \mathrm{visible}(\mathcal{O}, \mathsf{ctxt}) \cup \{\mathsf{null}\} \qquad \mathrm{type}_{\mathcal{O}}(j) = p.c \\ \langle m, \overline{q.t} \rightarrowtriangle \_, \_ \rangle \in_{KX} p.c \qquad \mathrm{mdecl}_{KX}(p.c, m) = \mathsf{comp} \qquad \overline{\mathrm{type}_{\mathcal{O}}(r) \leq_{KX} q.t} \qquad x \text{ fresh} \end{array}}{\mathcal{O}, \mathsf{nde} \dashrightarrow^{\mathsf{ctxt}}_{KX} \mathcal{O}, \mathsf{let}\ \mathit{lang.Object}\ x = j.m(\overline{r})\ \mathsf{in}\ \mathsf{nde}}$$

MGC-PrepareRtrn

$$\frac{r \in \mathrm{visible}(\mathcal{O}, \mathsf{ctxt}) \cup \{\mathsf{null}\} \qquad \mathrm{type}_{\mathcal{O}}(r) \leq_{KX} p.t}{KX, \mathcal{O}, \mathsf{nde}_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}} \overset{\tau}{\rightsquigarrow} KX, \mathcal{O}, r_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}}}$$

MGC-Fail

$$\frac{}{KX, \mathcal{O}, \mathsf{nde}_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}} \overset{\mathsf{error}}{\rightsquigarrow} KX, \mathcal{O}, \bullet}$$

Fig. 6. Transition rules for the most general context

expressions are only allowed in these most general contexts for LPJava components. Reducing a non-deterministic expression can lead to the creation of new objects, a well-formed cross-border method call / return using **RI-Call-Boundary / RI-Return-Boundary**) or abrupt program termination.

In order to distinguish contexts of the previous section from most general contexts, we call the previous ones *deterministic* contexts. Contexts then simply subsume both deterministic and most general contexts.

Construction of κ_X. The most general context of X is denoted by κ_X. Let mgc be a package name not occurring in X. For each abstracted type $T^\alpha = \langle \widehat{p.c}, \widehat{q.i}, \widehat{m} \rangle \in \mathcal{T}^\alpha_X$, we construct a class of the form:

---

package mgc; **public class** c **extends** $q.d$ **implements** $\overline{q.i}$ { $\widehat{M}$ }

---

where (1) c is a class name that is unique for each abstracted type T^α, (2) $q.d$ is the smallest class in $\widehat{p.c}$, (3) $\widehat{M}$ are the methods with signature from methods$_X(\widehat{p.c} \cup \widehat{q.i})$ which do not have names in $\widehat{m}$ and with nde as body. The idea behind this construction is that abstracting the type of the constructed class yields the abstracted type from which the class was constructed (i.e. typeabs$_{KX}(mgc.c) = T^\alpha$). The context κ_X also has an additional class Main, which is the startup class of $\kappa_X X$:

---

package main; **public class** Main { lang.Object main() { nde } }

Using this definition of most general context, we can give the denotation of a component X as the set of traces generated by the most general context of X. Note that this definition solely depends on X.

Definition 4 (Denotation of a component). *The denotation of a component X is defined as* Traces($\kappa_X X$).

3.3 Full Abstractness

The standard notion of testing or contextual compatibility [12] states that every program context which terminates with the first component must also terminate with the second component.

Definition 5 (Testing compatibility). *A component Y is* testing compatible *with X if Y is source compatible with X and for any deterministic program context K of X:* $\text{init}\zeta_{KX}\downarrow$ *implies* $\text{init}\zeta_{KY}\downarrow$.

The definition of testing compatibility quantifies over all possible program contexts and, as outlined for the challenges in the introduction, cannot be used in general for proving that two components are compatible. We therefore give an alternative definition that is based on the aforementioned denotations of components.

Definition 6 (Behavioral compatibility). *A component Y is* behaviorally compatible *with X if Y is source compatible with X and* $\text{Traces}(\kappa_X X) \subseteq \text{abs}_X(\text{Traces}(\kappa_Y Y))$.

We can not simply state trace inclusion, as Y may have more public types than X (see Def. 1). We must abstract from these additional types in the traces with the function $\text{abs}_X(T^\alpha)$ defined in Fig. 5. Finally we can state our main theorem, namely that the compatibility notions of Def. 5 and Def. 6 coincide.

Theorem 2 (Full abstraction). *Consider two components X and Y. Then Y is behaviorally compatible with X iff Y is testing compatible with X.*

The following lemmas are the main ones needed to prove full abstraction. Each of these are proven using simulation relations (see Section 4). The first two lemmas show that components and contexts compute the next label only based on the trace history, i.e., that the trace contains all relevant information.

Lemma 1 (Component independency). *Consider two contexts K_1 and K_2 for X such that* $t \in \text{Traces}(K_1X)$ *and* $t \in \text{Traces}(K_2X)$ *and* $\text{last}(t) = \mu?$. *Then* $t \cdot l \in \text{Traces}(K_1X)$ *implies* $t \cdot l \in \text{Traces}(K_2X)$.

Lemma 2 (Context independency). *Let Y be source compatible with X, K be a context for X and Y, and* $t \in \text{Traces}(KX)$ *and* $t \in \text{abs}_X(\text{Traces}(KY))$ *and* $t = \bullet$ *or* $\text{last}(t) = \mu!$. *Then,* $t \cdot l \in \text{Traces}(KX)$ *implies* $t \cdot l \in \text{abs}_X(\text{Traces}(KY))$.

The following two lemmas state that the most general context for a component X simulates exactly all possible contexts for X.

Lemma 3 (Trace abstraction). *Let K be a deterministic program context of component X. Then,* $\text{Traces}(KX) \subseteq \text{Traces}(\kappa_X X)$.

Lemma 4 (Trace concretization). *Let X be a component and* $t \in \text{Traces}(\kappa_X X)$. *Then, there is a deterministic program context K of X with* $t \in \text{Traces}(KX)$.

4 Simulations

In this section, we show how the main lemmas from the previous section can be proven using simulation relations on the runtime configurations. Before we can relate two configurations, we define in Section 4.1 a few well-formedness properties that the configurations must satisfy. We then define preorder relations over well-formed runtime configurations in Section 4.2. We show in Section 4.3 how these preorder relations are preserved by small operational steps. We also show that they are simulation relations on the large-step semantics in Section 4.4. Finally we describe how the main (trace-based) lemmas from the previous section can be proven using these simulation relations.

4.1 Well-Formed Runtime Configurations

Before giving the definition of well-formed runtime configuration, we first define a few helper functions. The function $\text{stackabs}_L(\overline{\mathcal{F}})$ yields all the L-tagged stack frames of $\overline{\mathcal{F}}$ and $\text{fieldrestrict}^L_{KX}(p.c,\overline{r})$ yields all field values of $\overline{r}$ that are defined in classes of L that are superclasses of $p.c$ or $p.c$ itself. The function $\text{objectrefs}(\ldots)$ yields all object identifiers contained in a syntactic element. We can then define well-formed runtime configurations.

Definition 7 (Well-formed runtime configuration). *A runtime configuration $\zeta = KX, \mathcal{O}, \overline{\mathcal{F}}$ is well-formed (denoted by $\mathsf{wf}(\zeta)$) if*

- *ζ is well-typed (standard definition, not detailed further here)*
- *Top of stack $\overline{\mathcal{F}}$ is an expression of the form $\mathcal{E}[E]$, the rest are contexts of the form $\mathcal{E}$*
- *Stack frames in $\overline{\mathcal{F}}$ are alternatively from* comp *and from* ctxt *and the lowest stack frame is from* ctxt
- *Store consistency:* $\text{objectrefs}(\text{rng}(\mathcal{O})) \subseteq \text{dom}(\mathcal{O})$
- *Stack consistency and separation:* $\forall L : \text{objectrefs}(\text{stackabs}_L(\overline{\mathcal{F}})) \subseteq \text{visible}(\mathcal{O}, L)$
- *Only L-visible objects can be accessed from L-visible objects:* $\forall j \in \text{visible}(\mathcal{O}, L)$ *with* $\mathcal{O}(j) = (\_,\_,p.c,\overline{r})$ *we have* $\text{objectrefs}(\text{fieldrestrict}^L_{KX}(p.c,\overline{r})) \subseteq \text{visible}(\mathcal{O}, L)$
- *Internal objects of X are of a type of X:* $\forall(\_,\mathsf{comp},p.c,\_) \in \text{rng}(\mathcal{O}) : p.c \in \mathcal{T}_X$
- *Internal objects of K have their X fields untouched:* $\forall(\mathsf{internal},\mathsf{ctxt},p.c,\overline{r}) \in \text{rng}(\mathcal{O}) : \text{fieldrestrict}^{\mathsf{comp}}_{KX}(p.c,\overline{r}) = \overline{\mathsf{null}}$

Initial program states are well-formed and well-formedness is preserved by small-step operational steps.

4.2 Preorder Relations $\preccurlyeq^\rho_L$

The preorder relations $\preccurlyeq^\rho_{\mathsf{comp}}$ and $\preccurlyeq^\rho_{\mathsf{ctxt}}$ relate two well-formed runtime configurations if their comp or ctxt part is similar. This allows us for example to relate runtime configurations when configurations only differ in the context or component (see e.g. Lemmas 1 and 2). We give their definition in the following. Note that helper functions are given in Fig. 5.

Definition 8 (Preorder relation $\preccurlyeq_L^\rho$). *Consider two well-formed configurations $\zeta_1 = K_1X_1, \mathcal{O}_1, \overline{\mathcal{F}_1}$ and $\zeta_2 = K_2X_2, \mathcal{O}_2, \overline{\mathcal{F}_2}$ such that X_2 is source compatible with X_1. We write $\zeta_1 \preccurlyeq_L^{\rho_e} \zeta_2$ if ρ_e is a renaming from* $\mathrm{filter}(\mathcal{O}_1, \mathsf{exposed})$ *to* $\mathrm{filter}(\mathcal{O}_2, \mathsf{exposed})$ *and there is a renaming ρ_i from* $\mathrm{filter}(\mathcal{O}_1, \mathsf{internal}, L)$ *to* $\mathrm{filter}(\mathcal{O}_2, \mathsf{internal}, L)$ *and $\rho = \rho_e \cup \rho_i$ such that*

– *if* $L = \mathsf{ctxt}$ *then* $K_1 = K_2$ *else* $X_1 = X_2$
– $\mathrm{currloc}(\zeta_1) = \mathrm{currloc}(\zeta_2)$
– $\mathrm{stackabs}_L(\overline{\mathcal{F}_1}) \equiv^\rho \mathrm{stackabs}_L(\overline{\mathcal{F}_2})$
– *If* $j_1 \equiv^\rho j_2$ *with* $\mathcal{O}_1(j_1) = (V_1, L_1, p_1.c_1, \overline{r_1})$ *and* $\mathcal{O}_2(j_2) = (V_2, L_2, p_2.c_2, \overline{r_2})$, *then*
  • $V_1 = V_2$
  • $L_1 = L_2$
  • $\mathrm{fieldrestrict}^L_{K_1X_1}(p_1.c_1, \overline{r_1}) \equiv^\rho \mathrm{fieldrestrict}^L_{K_2X_2}(p_2.c_2, \overline{r_2})$
  • *if* $L_1 = L$ *then* $p_1.c_1 = p_2.c_2$ *else* $\mathrm{typeabs}_{K_1X_1}(p_1.c_1) = \mathrm{abs}_{X_1}(\mathrm{typeabs}_{K_2X_2}(p_2.c_2))$

In the following, we explain the definition of $\preccurlyeq_L^\rho$. We first require that there is a renaming from the exposed objects of the first to the exposed objects of the second configuration. We also require that there is a renaming between the (internal) objects that are created by L. We then require that for both configurations the execution is at the same place (either in code of the component or the context). Furthermore, we require the parts of the stack that consist of code from L to be equal under the object renaming. For related objects, the heap entries must also match in the following way. The exposure and location flags must be the same. The values of fields that are defined in L must be equal under the object renaming. At last, the dynamic type of related objects must be equal if they are created by L. Otherwise, they must have the same abstracted types.

The relations $\preccurlyeq_L^\rho$ can be considered as simulation relations on the large-step semantics. We illustrate this in the following. Initial states are in the relation.

Lemma 5 (Initial states are related under $\preccurlyeq_{\mathsf{comp}}$). *Consider two program contexts K_1 and K_2 such that K_1X and K_2X are well-formed. Then* $\mathrm{init}\zeta_{K_1X} \preccurlyeq_{\mathsf{comp}} \mathrm{init}\zeta_{K_2X}$.

Lemma 6 (Initial states are related under $\preccurlyeq_{\mathsf{ctxt}}$). *Consider two components X_1 and X_2 such that X_2 is source compatible with X_1 and KX_1 and KX_2 are well-formed. Then* $\mathrm{init}\zeta_{KX_1} \preccurlyeq_{\mathsf{ctxt}} \mathrm{init}\zeta_{KX_2}$.

We first present how the relations $\preccurlyeq_L^\rho$ are preserved by steps of the small-step operational semantics and later extend it to the large-step one.

4.3 Small-Step Semantics

We consider four different cases. We distinguish whether the steps are labelled by τ or another label. We also distinguish whether the steps are initiated in the context or the component.

Lemma 7 (τ-steps in $\neg L$ preserve $\preccurlyeq_L$). *Assume that $\zeta_1 \preccurlyeq_L^\rho \zeta_2$ and* $\mathrm{currloc}(\zeta_1) = \neg L$. *If $\zeta_1 \overset{\tau}{\rightsquigarrow} \zeta_1'$ then $\zeta_1' \preccurlyeq_L^\rho \zeta_2$. Similarly, if $\zeta_2 \overset{\tau}{\rightsquigarrow} \zeta_2'$ then $\zeta_1 \preccurlyeq_L^\rho \zeta_2'$.*

Lemma 8 **($\preccurlyeq_L$ simulates τ-steps in L).** *If $\zeta_1 \preccurlyeq_L^\rho \zeta_2$ and $\zeta_1 \overset{\tau}{\rightsquigarrow} \zeta_1'$ and* currloc$(\zeta_1) = L$*, then $\zeta_2 \overset{\tau}{\rightsquigarrow} \zeta_2'$ and $\zeta_1' \preccurlyeq_L^\rho \zeta_2'$.*

In the following lemmas, we use the notion of *consistency* between renamings. Two renamings are consistent if the union of both relations yields a renaming again (i.e., they agree on the common value pairs).

Lemma 9 **(Similar messages from $\neg L$ preserve $\preccurlyeq_L$).** *If $\zeta_1 \preccurlyeq_L^\rho \zeta_2$ and $\zeta_1 \overset{l_1}{\rightsquigarrow} \zeta_1'$ and* currloc$(\zeta_1) = \neg L$ *and $\zeta_2 \overset{l_2}{\rightsquigarrow} \zeta_2'$ and $l_1 \equiv^{\rho_l}$* abs$_{X_1}(l_2) \neq \tau$ *and ρ_l minimal and consistent with ρ, then $\zeta_1' \preccurlyeq_L^{\rho \cup \rho_l} \zeta_2'$.*

Lemma 10 **($\preccurlyeq_L$ simulates messages from L).** *If $\zeta_1 \preccurlyeq_L^\rho \zeta_2$ and $\zeta_1 \overset{l_1}{\rightsquigarrow} \zeta_1'$ and* currloc$(\zeta_1) = L$ *and $l_1 \neq \tau$, then $\zeta_2 \overset{l_2}{\rightsquigarrow} \zeta_2'$ and $l_1 \equiv^{\rho_l}$* abs$_{X_1}(l_2)$ *and ρ_l minimal and consistent with ρ and $\zeta_1' \preccurlyeq_L^{\rho \cup \rho_l} \zeta_2'$.*

4.4 Large-Step Semantics

The four lemmas of the previous subsection can be extended to large steps and then to many large steps (i.e. program runs). For single large steps, we only state the lemma where a step is simulated.

Lemma 11 **($\preccurlyeq_L$ simulates large step from L).** *If $\zeta_1 \overset{l_1}{\Longrightarrow} \zeta_1'$ and* currloc$(\zeta_1) = L$ *and $\zeta_1 \preccurlyeq_L^\rho \zeta_2$, then $\zeta_2 \overset{l_2}{\Longrightarrow} \zeta_2'$ and $l_1 \equiv^{\rho_l}$* abs$_{X_1}(l_2)$ *and ρ_l minimal and consistent with ρ and $\zeta_1' \preccurlyeq_L^{\rho \cup \rho_l} \zeta_2'$.*

We then relate many large steps. For deterministic contexts, we can state that if we have a run starting from a state and another run starting from a related state which emits a similar trace as the first run, then the end states are related. We generalize this in the following lemma, where we also consider non-deterministic (i.e. most general) contexts.

Lemma 12 **(Multiple large steps preserve $\preccurlyeq_L$).** *If $\zeta_1 \overset{t_1}{\Longrightarrow} \zeta_1'$ and $\zeta_2 \overset{t_2}{\Longrightarrow} \zeta_2'$ and $\zeta_1 \preccurlyeq_L^\rho \zeta_2$ and $t_1 \equiv^{\rho_t^{12}}$* abs$_{X_1}(t_2)$ *and ρ_t^{12} minimal and consistent with ρ, then $\exists \zeta_3'$ such that $\zeta_2 \overset{t_3}{\Longrightarrow} \zeta_3'$ and $t_1 \equiv^{\rho_t^{13}}$* abs$_{X_1}(t_3)$ *and ρ_t^{13} minimal and consistent with ρ and $\zeta_1' \preccurlyeq_L^{\rho \cup \rho_t^{13}} \zeta_3'$.*

The states ζ_1' and ζ_2' might not be related, as during the runs $\zeta_1 \overset{t_1}{\Longrightarrow} \zeta_1'$ and $\zeta_2 \overset{t_2}{\Longrightarrow} \zeta_2'$, the same most general context might have chosen different executions as it is non-deterministic. For example, it may create more objects that are internal to it in one execution than in another, but still generate a similar trace. For deterministic contexts, however, $\zeta_1' \preccurlyeq_L \zeta_2'$.

We can finally prove Lemmas 1 and 2. We know that initial states are related and that after the traces are executed, the states thereafter are still related (by Lemma 12). By

Lemma 11, we then know that the second configuration can respond in a similar way to the first one.

To prove Lemmas 3 and 4, however, we need stronger relations that not only relate the comp part of the configurations, but also relate the most general context to the deterministic context (which we denote by $\lll^{\rho}$ for trace abstraction and $\ggg^{\rho}$ for trace concretization). The proof then works in a similar way, where similar lemmas as before have to be proven for the relations $\preccurlyeq^{\rho}_{\mathsf{comp}} \cap \lll^{\rho}$ and $\preccurlyeq^{\rho}_{\mathsf{comp}} \cap \ggg^{\rho}$.

5 Proving Compatibility

In this section, we give proof obligations that are needed in order to prove two components compatible. We also describe why the proof method is complete and sketch how the direct connection of the trace semantics to the operational semantics can be exploited to prove compatibility.

In order to prove that a component Y is behaviorally compatible with X, the following steps are necessary. First, Y must be proven source compatible with X. This can be directly done by the checks detailed in Section 2. The more difficult part is to prove, as per Def. 6, that $\text{Traces}(\kappa_X X) \subseteq \text{abs}_X(\text{Traces}(\kappa_Y Y))$. It is sufficient to prove that $\text{Traces}(\kappa_X X) \subseteq \text{abs}_X(\text{Traces}(\kappa_X Y))$, which follows from the property that $\text{Traces}(\kappa_X Y) \subseteq \text{Traces}(\kappa_Y Y)$.

The proof is done by induction on the length of the traces. The empty trace is trivially in both sets. As induction step, assume $t \cdot l$ such that (1) $t \in \text{Traces}(\kappa_X X)$, (2) $t \in \text{abs}_X(\text{Traces}(\kappa_X Y))$, and (3) $t \cdot l \in \text{Traces}(\kappa_X X)$. The proof goal is then to show that $t \cdot l \in \text{abs}_X(\text{Traces}(\kappa_X Y))$. We distinguish two cases, based on the form of the last label in the trace t:

Case $t = \bullet$ or $\text{last}(t) = \mu!$: The claim follows directly by Lemma 2.

Case $\text{last}(t) = \mu?$: The initial configurations of $\kappa_X X$ and $\kappa_X Y$ are related by $\preccurlyeq_{\mathsf{ctxt}}$ due to Lemma 6. By Lemma 12, the configurations right after the trace t are related by $\preccurlyeq_{\mathsf{ctxt}}$ as well. It then suffices to prove that the second configuration can run and generate a next label whenever the first configuration runs and generates this label. There are different approaches for proving this. One possibility is to capture how the comp part of both configurations are related. This relation, usually called *coupling invariant* [3], only needs to relate the comp parts of the configuration. Note that these parts remain untouched by the context during its steps. Another possibility is to relate the runtime configurations to the traces, i.e., for each component X establish a relation between the traces of $\text{Traces}(\kappa_X X)$ and runtime configurations of $\kappa_X X$ (which we call a *canonical representation invariant*). We consider the description of more detailed approaches as future work.

Completeness of the Proof Method. Although we do not consider in this paper how a coupling invariant can be specified, we can show that such an invariant always exists if two components are behaviorally compatible. This completeness result comes basically for free from our full abstraction proof approach. If two components are behaviorally compatible, then there always exists a coupling invariant (simulation relation) $\mathcal{R}$ which can roughly be constructed as follows:

– $(\mathrm{init}\zeta_{\kappa_X X}, \mathrm{init}\zeta_{\kappa_X Y}) \in \mathcal{R}$.
– If $\mathrm{init}\zeta_{\kappa_X X} \xRightarrow{t_1} \zeta_1$ and $\mathrm{init}\zeta_{\kappa_X Y} \xRightarrow{t_2} \zeta_2$ and $t_1 \equiv t_2$, then $(\zeta_1, \zeta_2) \in \mathcal{R}$.

We know already that for each $(\zeta_1, \zeta_2) \in \mathcal{R} : \zeta_1 \preccurlyeq^{\rho}_{\mathsf{ctxt}} \zeta_2$. We also know that the ctxt part is left untouched when the component executes (and the execution is independent of the context). Thus, a user-specified invariant only needs to talk about the comp part of the configuration. If the invariant only talks about the comp part, then we also have the guarantee that it remains untouched by the context, which allows us to disregard steps in the (most general) context.

Weaker Compatibilites. Sometimes we are not interested in preserving the full behavior of components in an evolution step. For example, we might be interested in checking that the new version of a component has the same behavior as the old one with respect to a subset of its interface methods. In this case, we can compare only those traces including these methods. This is a typical evolution scenario. Similarly, the approach can be adapted to prove that components behave the same in a restricted set of contexts. Weaker compatibilities can be considered for example by (1) providing a more restrictive definition of the most general context (e.g. disallow the context to call a certain method) (2) giving an abstraction function on the traces or (3) specifying method contracts that must be satisfied by contexts.

6 Conclusion and Future Work

We have presented a fully abstract trace-based semantics for packages of an object-oriented class-based language. We have also the shown the relation of the trace-based to the operational semantics and provided a proof outline of the full abstraction result where simulation relations are used. We see this as foundational work for relating trace-based specifications to components (i.e. sets of classes), proving sets of classes compatible or equivalent and proving refactoring transformations behavior preserving.

A particular reason for why the trace-based approach fits so well is that object-oriented programming is based on message passing. We are not sure whether deriving fully abstract semantics using the mixed trace-based/operational semantics approach fits well for other non-OO settings, but plan on exploring this in the future. Furthermore, we would like to work out a concrete proof technique which consists of providing a specification language for describing coupling relations, extending existing program logics for OO programs [1,14] to our setting and automatically generating proof obligations to be used by interactive or automatic theorem provers.

Acknowledgements. We thank Christoph Feller and Jean-Marie Gaillourdet for commentating on earlier drafts of this paper. We also thank the anonymous reviewers of SBMF 2011 for their valuable feedback.

References

1. Abadi, M., Leino, K.R.M.: A logic of object-oriented programs. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 11–41. Springer, Heidelberg (2004)

2. Ábrahám, E., Bonsangue, M.M., de Boer, F.S., Steffen, M.: Object connectivity and full abstraction for a concurrent calculus of classes. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 37–51. Springer, Heidelberg (2005)
3. Back, R.J.J., Akademi, A., Wright, J.V.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (1998)
4. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. Journal of the ACM 52(6), 894–960 (2005)
5. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. The Java Series. Addison-Wesley, Boston (2005)
6. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: OOPSLA, pp. 241–253 (2001)
7. Hennessy, M., Milner, R.: On observing nondeterminism and concurrency. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 299–309. Springer, Heidelberg (1980)
8. Jeffrey, A., Rathke, J.: Java Jr.: Fully abstract trace semantics for a core Java language. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 423–438. Springer, Heidelberg (2005)
9. Koutavas, V., Wand, M.: Bisimulations for untyped imperative objects. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 146–161. Springer, Heidelberg (2006)
10. Koutavas, V., Wand, M.: Reasoning about class behavior. In: Informal Workshop Record of FOOL 2007 (January 2007)
11. Milner, R.: Fully abstract models of typed lambda-calculi. Theor. Comput. Sci. 4(1), 1–22 (1977)
12. Morris, J.H.: Lambda-calculus models of programming languages. Tech. Rep. 57, MIT Laboratory for Computer Science (1968)
13. Plotkin, G.D.: Lcf considered as a programming language. Theor. Comput. Sci. 5(3), 223–255 (1977)
14. Poetzsch-Heffter, A., Müller, P.: A programming logic for sequential java. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 162–176. Springer, Heidelberg (1999)
15. Silva, L., Naumann, D.A., Sampaio, A.: Refactoring and representation independence for class hierarchies: Extended abstract. In: Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs, FTFJP 2010, pp. 8:1–8:7. ACM, New York (2010)
16. Sumii, E., Pierce, B.C.: A bisimulation for dynamic sealing. Theoretical Computer Science 375 (2007)
17. Sumii, E., Pierce, B.C.: A bisimulation for type abstraction and recursion. Journal of the ACM 54 (2007)
18. Welsch, Y., Poetzsch-Heffter, A.: Full abstraction at package boundaries of object-oriented languages. Tech. Rep. 384/11 (May 2011), http://softech.cs.uni-kl.de/Homepage/PublikationsDetail?id=157
19. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. 115(1), 38–94 (1994)

B to CSP Migration: Towards a Formal and Automated Model-Driven Engineering of Hardware/Software Co-design

Marcel Vinicius Medeiros Oliveira, David B.P. Déharbe, and Luís C.D.S. Cruz[*]

Universidade Federal do Rio Grande do Norte – Brazil
marcel@dimap.ufrn.br, deharbe@gmail.com

Abstract. This paper presents a migration approach from a class of hierarchical B models to CSP. The B models follow a so-called *polling pattern*, suitable for reactive systems, and are automatically translated into a set of communicating CSP processes with the same behaviour. The structure of the CSP model matches that of the B model and may be formally analysed using model checking. Selected CSP processes may then be further refined and synthesised to hardware, while the remaining modules would be mapped to software using B refinements. The translation proposed here paves the way for a model-based approach to hardware and software co-design employing complementary formal methods.

Keywords: Model-driven engineering, co-design, B, CSP.

1 Introduction

Model-driven engineering is based on models where software and hardware are built by designing, analysing, and transforming models [10]. Model-driven development is performed in formalisms that support transformations between different levels of abstractions (e.g. refinement and code synthesis) or at the same level of abstraction (e.g. refactoring and translation). In the case of *embedded systems*, several interacting components may be either mapped to software (i.e. running on a micro-controller) or to hardware (either a custom design or on a programmable artifact such as a FPGA). Hence, model-based design of embedded systems necessitates formalisms that can be mapped to either software or to hardware and that support analysis to inspect properties about concurrency, communication as well as data consistency and program termination.

Languages such as Z [17] and B [1] are specification languages suitable for a model-based approach to specification. In these languages, modelling aspects such as concurrency and communication is possible, but awkward and difficult as they do not have specific language constructs for concurrency. Process algebras such as CSP [8] and CCS [11] provide constructs to describe dynamic behaviour

[*] INES and CNPq partially supports the work of the authors: grants 573964/2008-4, 476836/2009-3, 560014/2010-4, and 306033/2009-7.

but lack concise ways to describe complex data. So there has been many attempts to combine these two kinds of formalism: Z with CCS [16], Z with CSP [14], Object-Z with CSP [6], and *Circus* [12] are some examples.

One of our industrial partners uses the B method to develop reactive control systems; changing formalism is out of question for strategic reasons. In order to provide for a broader analysis, we propose a migration path from B to CSP, supported by a translator that checks restrictions on the B model and generates a CSP model using the transformation discussed herein. In [4], we define a pattern to model reactive systems in B. By automatically translating these models to CSP, we may check the safety requirements of the B model on the resulting CSP model as well as absence of deadlock and livelock using tools such as FDR [7].

In [13], we present a translation from CSP to Handel-C, a high level programming language which targets low-level hardware, most specifically FPGA programming. So, in addition to a broader analysis of the system properties, the proposed migration also permits targeting FPGAs. This work thus provides the elements of a model-based approach that, starting from a unique formalism, B, may be used to verify static and dynamic behavioural properties, as well as building both software and hardware components.

Migration between formalisms is not new: some were motivated by the same reasons as ours and others were used to give semantics of combinations of different formalisms. Butler and Waldén present an embedding of action systems in the B-method in [2], where they also compare the refinement notions of action systems and B, and suggest extensions. Butler has used these results as an inspiration for csp2b [3], a tool to translate a subset of CSP into B. In [14], Woodcock *et al.* give an informal translation from Z to CSP by separating input and output communications. Fischer generalises this result in his work with CSP-OZ [6], a combination of Object-Z and CSP. Finally, [5] describes the notation CSP||B that allows to describe dynamic behaviour explicitly in a B-like model. As far as we know, no previous work provides a migration of models from B to CSP.

This paper is organised as follows. In Section 2 we propose a methodology for software-hardware co-design based on B, CSP and the translation strategy proposed in this paper. Section 3 describes the case study that we use here to illustrate the translation strategy presented in Section 4. Finally, we conclude the paper with some further discussions on our results and future work.

2 Methodology

We propose a methodology (depicted in Figure 1) to model check and generate code from B specifications that obey a so-called *polling pattern*. This pattern provides a mean to model reactive systems in B. The B model has a single operation that models the update to the outputs corresponding to a change to the inputs. The state of the model includes the values of the inputs, outputs and possibly some internal variables. Hence the state invariant may also specify constraints on the system inputs and outputs. Following the B method, the abstract B model is refined using tools such as Atelier B. We propose an automatic

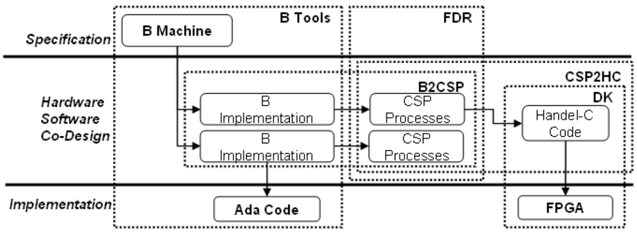


Fig. 1. Methodology Overview

translation to CSP and the use of FDR to model check properties like absence of deadlock and livelock in the system. Throughout this paper, we assume the reader is familiarised B and CSP.

Specification modules may target either software or hardware. The former can be automatically generated using existing code generators for B. The latter can be generated using the translator presented in [13], which translates CSP into Handel-C, a programming language similar to C, used to program FPGAs.

The transformation rules are based on a pattern of B specifications for reactive systems, derived from [4], with a few syntactic adaptations to facilitate the definitions of these rules. There are two patterns: one for unique machines (e.g. the conduction mode definition component in Figure 2), and one for replicated machines (e.g. the side door controller). The latter is similar to the former, but takes into account that, when instantiating machines, their variable names are changed by prefixing them with a unique name.

The unique machines access machines defining auxiliary constants and sets; they have a number of inputs (inp_i), outputs (out_i) and internal variables (var_i); the invariant defines their types (*type*), assumptions about inputs from the environment (*env_ass*), and safety conditions (*safety*). In the pattern, every variable of a machine *PP* is initialised with a given value and there exists a single operation, *updt_PP*, that receives input values as arguments; its precondition specifies the type of inputs and assumptions about the environment. This operation updates, in parallel, the values of the input and output variables. The new values of the output and internal variables are given by a transition function taking as arguments the current values of the input and internal variables.

3 A Case Study

The automatic translation from (a subset of) B to CSP is the core of the methodology; it is illustrated on a subway control system responsible for managing the opening and the closing of the car doors [4]. The full system is comprised of one controller per door, one controller per car, and one controller for each pilot cabin (there is one cabin at each end of the train). The case study is named *general door controller* (GDC) and is the subsystem found in each pilot cabin.

```
MACHINE $PP$
SEES $M_0, \ldots, M_j$
VARIABLES $inp_0, \ldots, out_0, \ldots, var_0, \ldots$
INVARIANT
  $type(inp_0) \wedge \ldots\ type(out_0) \wedge \ldots\ type(var_0) \wedge \ldots\ \wedge$
  $env\_ass(inp_0, \ldots) \wedge safety(inp_0, \ldots, out_0, \ldots, var_0, \ldots)$
INITIALISATION $inp_0 := i_0 \,||\, \ldots \,||\, out_0 := o_0 \,||\, \ldots \,||\, var_0 := v_0$
OPERATIONS
  $updt\_PP(i\_inp_0, \ldots, i\_inp_n) =$
    PRE $type(i\_inp_0) \wedge \ldots \wedge env\_ass(i\_inp_0, \ldots)$
    THEN $inp_0 := i\_inp_0 \,||\, \ldots \,||\, trans\_o_0(i\_inp_0, \ldots, var_0, \ldots) \,||\, \ldots \,||$
         $trans\_var_0(i\_inp_0, \ldots, var_0, \ldots) \,||\, \ldots$
    END
END
```

Fig. 2. Polling Pattern (unique)

The interface and internal structure of the GDC are schematised in Figure 3. The GDC handles doors on both sides of the train. Replicated components (for the left and right sides) are depicted once with bold lines. The GDC is responsible for controlling four signals:

o_traction , when false, interrupts the wheel traction. It is used for safety emergencies, e.g. when doors are open.
o_cmd commands the opening and closing of all the doors.
o_visual commands a visual signal indicating a door is open.
o_oslow commands a visual signal indicating the speed is low.

The GDC shall set its outputs only when its cabin is the train *leader*. Indeed, there are two cabins, but only one of them may be the leader at a given time. The GDC has an internal variable *c_cabin* that records if the leader is its own cabin, the opposite cabin, or none of them. Its value is updated by the component **leader definition**, based on two inputs: *i_ldr* and *i_opp*.

The GDC has three conduction modes: manual, controlled, and off. The variable *c_mode* stores its current mode; its value is determined by the component **mode definition** based on two inputs that are driven by a switch in the cabin. The values of *c_cabin* and *c_mode* are combined by the component **condition** into a value *c_cond* used to determine whether the cabin shall control the doors.

There are two instances of the subsystem **side door controller**, for the left and right sides respectively. They control the *o_cmd* and *o_visual* outputs, based on the value of the inputs *i_open* and *i_close*, driven by a lever controlled by the pilot, *i_sensor* driven by a door sensor, and the internal signal *c_cond* described previously. The subsystems **traction** and **slow** drive the corresponding output signals, based on inputs *i_sensor* and *i_slow*, respectively, which are themselves driven by hardware sensors.

Due to space restrictions, only part of the B model is presented. The different subsystems are modelled using a template for specifying reactive systems using

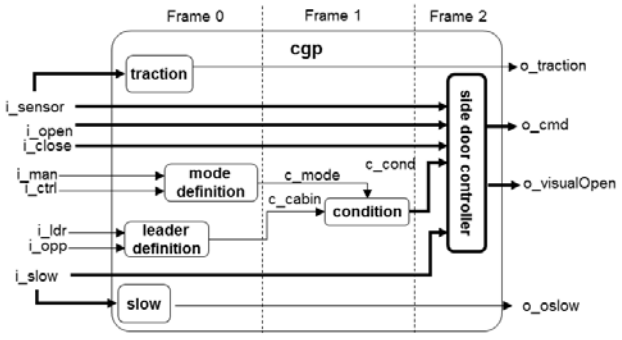


Fig. 3. General Door Controller

the B notation set forth in [4]. Figure 4 provides an instance of this pattern in the subsystem **mode definition**. The model includes state variables to represent system input, state and output and a single operation that has as parameters the new values of the subsystem inputs and models an update. The variables representing the inputs are updated with their new values. The body of the operation is a parallel update of each model variable. The model for the full system is composed of nine modules with a total number of 350 lines. Module *CONTEXT* contains type and function definitions.

The full model has been verified using Atelier-B [1]. Next, we present the transformation rules to translate this machine (and the others) into a CSP specification that can be verified using FDR, including additional properties such as livelock and deadlock-freedom.

4 Transformation of B Specifications

Our translation from B to CSP allows an automatic re-check of safety requirements from the original B model; it yields a CSP model that contains: (1) generic auxiliary processes; (2) CSP paragraphs (channel declarations, initial values, alphabets, assumptions, safety specifications, auxiliary functions, and module behaviour) that correspond to each B machine; and (3) a controller that synchronizes all processes before inputs and outputs. Every produced CSP specification contains the following CSP processes: `RUN(A)` always offers events from set `A`; `CHAOS(A)` behaves similarly, but may deadlock at any time; `DIV` diverges. All definitions can be found at `http://is.gd/B2CSPall` along with an implementation of the translator.

Figure 3 depicts the dependencies of the GDC components. For instance, component *condition* depends on *mode definition* and *leader definition*. Hence, *condition* must be executed after *mode definition* and *leader definition*. Consider

[1] http://www.atelierb.eu

MACHINE $\mathit{CONTEXT}$
SETS $\mathit{COND} = \{\mathit{MAN}, \mathit{CTRL}, \mathit{COND\_OFF}\}$... **END**

MACHINE $\mathit{CMODE\_DEF}$
SEES $\mathit{CONTEXT}$
VARIABLES $\mathit{i\_man}, \mathit{i\_ctrl}, \mathit{c\_mode}$
INVARIANT
$\mathit{i\_man} \in \mathit{BOOL} \wedge \mathit{i\_ctrl} \in \mathit{BOOL} \wedge \mathit{c\_mode} \in \mathit{COND}$
$\wedge\ ((\mathit{i\_man} = \mathit{TRUE} \wedge \mathit{i\_ctrl} = \mathit{FALSE}) \Rightarrow \mathit{c\_mode} = \mathit{MAN})$
$\wedge\ ((\mathit{i\_man} = \mathit{FALSE} \wedge \mathit{i\_ctrl} = \mathit{TRUE}) \Rightarrow \mathit{c\_mode} = \mathit{CTRL})$
$\wedge\ (((\mathit{i\_man} = \mathit{FALSE} \wedge \mathit{i\_ctrl} = \mathit{FALSE}) \vee (\mathit{i\_man} = \mathit{TRUE} \wedge \mathit{i\_ctrl} = \mathit{TRUE})) \Rightarrow$
$\quad \mathit{c\_mode} = \mathit{COND\_OFF})$
INITIALISATION $\mathit{i\_man} := \mathit{TRUE} \parallel \mathit{i\_ctrl} := \mathit{FALSE} \parallel \mathit{c\_mode} := \mathit{MAN}$
OPERATIONS
$\mathit{updt\_CMODE\_DEF}(\mathit{a\_man}, \mathit{a\_ctrl}) =$
  PRE $\mathit{a\_man} \in \mathit{BOOL} \wedge \mathit{a\_ctrl} \in \mathit{BOOL}$ **THEN**
  $\mathit{i\_man} := \mathit{a\_man} \parallel \mathit{i\_ctrl} := \mathit{a\_ctrl} \parallel$
  $\mathit{c\_mode} : (\mathit{c\_mode} : \mathit{CONDUCTION} \wedge$
  $\qquad (\mathit{a\_man} = \mathit{TRUE} \wedge \mathit{a\_ctrl} = \mathit{TRUE} \Rightarrow \mathit{c\_mode} = \mathit{COND\_OFF}) \wedge$
  $\qquad (\mathit{a\_man} = \mathit{TRUE} \wedge \mathit{a\_ctrl} = \mathit{FALSE} \Rightarrow \mathit{c\_mode} = \mathit{MAN}) \wedge$
  $\qquad (\mathit{a\_man} = \mathit{FALSE} \wedge \mathit{a\_ctrl} = \mathit{TRUE} \Rightarrow \mathit{c\_mode} = \mathit{CTRL}) \wedge$
  $\qquad (\mathit{a\_man} = \mathit{FALSE} \wedge \mathit{a\_ctrl} = \mathit{FALSE} \Rightarrow \mathit{c\_mode} = \mathit{COND\_OFF}))$
  END
END

Fig. 4. Definition of the conduction mode machine

the corresponding dependency graph. Since mutual dependencies are disallowed it is a directed acyclic graph (DAG). The DAG determines the scheduling of the components as follows. Each component is allocated to an *execution* frame corresponding to its level in the DAG. A process `CONTROL` coordinates the processes representing B machines using channels `read` and `write`. It guarantees that the execution order respects the dependencies. When all frames have been executed, `CONTROL` signals the end of the cycle and recurses.

```
datatype FRAME = F0 | F1 | ...
EXEC_FRAME = <F0, F1, ...>
channel read, write:FRAME
channel end_cycle
A_CONTROL_ALL = Union({A_CONTROL(i) | i <- FRAME})
A_CONTROL(i) = {| read.i, write.i, end_cycle |}
CONTROL = (; i:EXEC_FRAME @ read.i -> write.i -> SKIP);
          end_cycle -> CONTROL
```

In B, the machine `CONTEXT` defines the types used in the system:

MACHINE $\mathit{CONTEXT}$ **SETS** $S_0 = \{a_0, \ldots\}, \ldots$ **END**

Table 1. CSP Paragraphs for B Components

Type	Description
Channels	input and output channels corresponding to the machine input and output variables
Initial Values	initial values for input and local variables
Alphabet	set of events used within the process
Assumptions	predicate corresponding to the B environment assumption about the input values
Safety	process specifying the safety conditions
Functions	functions computing and checking each output
Process	processes representing the B machine

The translation of such enumerated set in CSP is:

```
datatype S_0 = a_0 | ... | a_n | S_0_DUMMY
```

The translation of some expressions may lead to an undetermined value. For these cases we introduce a DUMMY value. However, the translation inserts assertions that guarantee that such values never appear in the traces. In our example, the translation yields the data type:

```
datatype COND = MAN | CTRL | COND_OFF | COND_DUMMY
```

For each B machine (except the machine that represents the whole system), the translation yields the CSP paragraphs presented in Table 1. In addition to these definitions, we have the specification of the overall system and CSP assertions that are used in the verification process. In what follows, we describe each one of these paragraphs and illustrate it with our example. Throughout the translation, we assume that name clashes have been removed with a simple pre-processing of the original B specification.

Channels that correspond to each input and output variable are declared using the types given to variables in the machine invariant. We consider a function B2CSP that transforms B expressions into CSP expressions. The expressions currently translated are type expressions (*i.e* integer and boolean), constants, assignments, and arithmetic and boolean expressions. The definition of B2CSP is relatively simple, but lengthy. For conciseness, it is omitted here.

```
channel i_inp_0 : B2CSP(type(inp_0)) ...
channel o_out_0 : B2CSP(type(out_0)) ...
```

Our example has two boolean inputs and one output of type COND.

```
channel i_man  : Bool
channel i_ctrl : Bool
channel c_mode : COND
```

Initialisation of the B machines define the initial values of each variable.

```
I_inp_0 = B2CSP(i_n) ...
I_out_0 = B2CSP(o_0) ...
I_var_0 = B2CSP(v_0) ...
```

Here, `man`, `ctrl`, and `mode` are initialised with `TRUE`, `FALSE`, and `MAN`, respectively.

```
I_man  = true
I_ctrl = false
I_mode = MAN
```

The **Alphabet** of each process is the union of the controller's alphabet with the corresponding machine's channels.

```
A_PP(frame) = union(A_CONTROL(frame),{|i_inp_0, ..., o_out_0, ...|})
```

In our case, we have the following alphabet.

```
A_CMODE_DEF(frame) = union(A_CONTROL(frame), {| i_man, i_ctrl, c_mode |})
```

Input assumption is a formula on inputs, it is handled with function `B2CSP`.

```
PP_inp_assump(inp_0, ...) = B2CSP(env_assump(i_inp_0, ...))
```

Since the example makes no explicit assumptions on the inputs, the function `CMODE_DEF_inp_assump` yields `true`.

Safety conditions describe the relations between input, local, and output variables. We assume they are as follows:

$$safety(inp_0, \ldots, out_0, \ldots, var_0, \ldots) = In_{0_0}(inp_0, \ldots, var_0, \ldots) \Rightarrow Out_{0_0}(out_0) \wedge \ldots$$

For each output, a predicate gives the conditions under which it must be assigned some expression. These predicates are composed in a conjunction. Such safety conditions are translated to a process that reads the inputs and restricts the outputs only if the input values satisfy the assumptions given in the pre-condition of the B operation. In these cases, each conjunct in the safety condition yields a choice in the process: the left-hand side guards the offer of the output of a value described by the right-hand side. Finally, an extra choice that does not restrict the output handles the case in which no left-hand side condition is satisfied.

```
PP_safe = i_inp_0?inp_0 -> ... -> i_inp_n?inp_n ->
        ( if (PP_inp_assump(inp_0, ...))
          then ((B2CSP(In_0_0(inp_0, ..., var_0, ...)) &
                    o_out_0!B2CSP(Out_0_0(out_0)) -> SKIP) [] ...
                 [] (((not In_0_0(...)) and ... )) & o_out_0?out_0 -> SKIP)
          else ( o_out_0?out_0 -> SKIP ) ); PP_safe
```

The conduction mode is `MAN` if and only if inputs `man` and `ctrl` are `TRUE` and `FALSE`; `CTRL` if they are `FALSE` and `TRUE`; `COND_OFF` if they are equal.

```
CMODE_DEF_safe =
  i_man?man -> i_ctrl?ctrl ->
  ( if (CMODE_DEF_inp_assump(man, ctrl))
    then ( ((man) and (not ctrl)) & c_mode!MAN -> SKIP
         []((not man) and (ctrl)) & c_mode!CTRL -> SKIP
         [](((not man) and (not ctrl)) or ((man) and (ctrl))) &
            c_mode!COND_OFF -> SKIP
         []((not((man) and (not ctrl))) and (not((not man) and (ctrl))) and
            (not(((not man) and (not ctrl)) or ((man) and (ctrl))))) &
            c_mode?mode -> SKIP )
    else (c_mode?mode -> SKIP) ); CMODE_DEF_safe
```

In this process, after reading the inputs, the output on `c_mode` is restricted only if the input values satisfy the assumptions: if `man` is `true` and `ctrl` is `false`, `MAN` is communicated, and similarly for the other output values. Finally, if no condition is satisfied, any value is output.

Output functions calculate output values based on the values of the local and input variables. The B model specifies the outputs as expressions on inputs and state variables which are translated with function `B2CSP`. For each B output `out_i`, the translation yields the following CSP function:

```
PP_trans_out_i(var_0, ..., inp_0, ...) =
    B2CSP(trans_out_i(var_0, ..., inp_0, ...))
```

The example has a single output, `mode`, calculated by the following CSP function, translated from the expression used in Figure 4.

```
CMODE_DEF_trans_mode(man, ctrl) =
    if ((man) and (ctrl)) then COND_OFF else
    if ((man) and (not ctrl)) then MAN else
    if ((not man) and (ctrl)) then CTRL else
    if ((not man) and (not ctrl)) then COND_OFF else COND_DUMMY
```

Similar functions are defined for the transition of the local variables. Our example has no local variables (but the full specification does have local variables). Here, we also have the definition of `CMODE_DEF_safe_trans(man, ctrl)`, which checks that no transitions produce a dummy value. We now turn to the production of the processes that correspond to the machine itself.

Four CSP **Processes** are used to describe the behaviour of each system component `PP`. Each of these processes corresponds to a stage in the machine life-cycle: initialisation, input, internal communication, and output. Furthermore, each machine (and its corresponding CSP processes) must be allocated into one of the execution frames.

The initialisation process uses the initial values of the local variables to start the process that reads the external inputs.

```
PP(frame) = PP_INPUT(frame, I_var_0, ...)
```

As the example has no local variables, there is no arguments other than that used to indicate the component's frame:

```
CMODE_DEF(frame) = CMODE_DEF_INPUT(frame)
```

The next process reads all the inputs that are given by the external environment. Nevertheless, it must first synchronise with the other processes and the controller on the `read` event to prevent it from reading inputs outside its execution frame. The communicated values and the initial values of local variables are used to instantiate the next process that reads internal communications.

```
PP_INPUT(frame, var_0, ..., var_k) =
    read.frame -> i_inp_0?inp_0 -> ... PP_INTERNAL(var_0, ..., inp_0,...)
```

The **mode definition** component reads two inputs:

```
CMODE_DEF_INPUT(frame) =
    read.frame -> i_man?man -> i_ctrl?ctrl ->
    CMODE_DEF_INTERNAL(frame, man, ctrl)
```

After reading the environment inputs, the process reads internal communications that are input to the corresponding B machine. In our naming convention, these are the components prefixed with `c_` such as `c_mode`. This process synchronises with the other processes on the `write` event before receiving values given by other processes.

```
PP_INTERNAL(frame, var_0, ..., var_k, inp_0,...) =
 write.frame -> c_inp_n?inp_n -> ...
   if not(PP_inp_assump(inp_0,...) and PP_safe_trans(inp_0,...)) then DIV
   else PP_OUTPUT(frame, var_0, ..., inp_0,...)
```

If the transition is safe (i.e. there are no dummy values), the process writes the outputs; it diverges otherwise.

The example has a single output, `c_mode`. For this reason, at this stage we have no further communication. The process `CMODE_DEF_INTERNAL` simply synchronises on `write` and behaves like `CMODE_DEF_OUTPUT` described below.

```
CMODE_DEF_INTERNAL(frame, man, controlled) =
  write.frame ->
  if not(CMODE_DEF_safe_trans(man, ctrl))
  then DIV else CMODE_DEF_OUTPUT(frame, man, ctrl)
```

The last process that describes the behaviour of the machine writes its outputs using the transition functions. Finally, it waits for the end of the cycle and transitions back to the input stage with the local variables updated according to the transition function:

```
PP_OUTPUT(frame, var_0, ..., inp_0, ...) =
  o_out_0!PP_trans_out_0(var_0, ..., inp_0, ...) -> ... -> end_cycle ->
  PP_INPUT(frame, PP_trans_var_0(var_0, ..., inp_0, ...), ...)
```

The example has no local variables, the translation yields an output on `c_mode`, using the transition function followed by input process `CMODE_DEF_INPUT`.

```
CMODE_DEF_OUTPUT(frame, man, ctrl) =
  c_mode!CMODE_DEF_trans_mode(man, ctrl) -> end_cycle ->
  CMODE_DEF_INPUT(frame)
```

This concludes the translation of component machines such as the conduction mode. The translation of the leader definition module is very similar. The translation of the remaining components takes into consideration the replication of one of the GDC's components, the side door controller, using B instantiation. In this case, we must apply syntactic changes, described below, to the result of the translation presented so far.

4.1 Systems with Replicated Machines

Replicated machines are those that are replicated using B instantiation. For instance, the machine modelling the general door controller includes two instances of the side door controller named `Left` and `Right` (see Figure 5).

We consider an environment that stores the instances of each replicated machine, if any. For instance, if the main system includes the machines `Inst_0.PP`, ..., `Inst_p.PP`, we have the environment $\mathcal{S} \mathrel{\widehat=} \{ PP \mapsto \{Inst_0, \ldots, Inst_p\} \}$, and the resulting CSP specification includes a data type that represents the possible instances of a given machine:

```
datatype PP_symm = Inst_0 | ... | Inst_p
```

The following data type corresponds to the replication of the side door controller in the GDC.

```
datatype S_DOOR_CTRL_symm = Left | Right
```

In the presence of replicated machines, syntactic changes are added to the translation presented so far. There are three kinds of changes: global changes, changes for unique machines and changes for replicated machines.

Global Changes alter the declaration of the channels that correspond to components of replicated machines. For every variable `v` declared in the **VARIABLES** block of a replicated machine `PP`, the type `PP_symm` becomes the first type in the channel declaration that corresponds to `v`. For instance, `c_cond` should be declared simply as a boolean channel. However, since `cond` is an input of the side door controller (see Figure 3), its corresponding channel is declared as:

```
channel c_cond: S_DOOR_CTRL_symm.Bool
```

The change in the channel type is reflected in the translation of all paragraphs (see Table 1) of the translated machines.

Unique Machines For every variable `v` in the **VARIABLES** block of a replicated machine `PP`, and labels `Inst_`i for its instances, variable names `name_Inst_`i are substituted with `name.Inst_`i. For example, `c_cond_Left` is substituted by `c_cond.Left` in the processes for machine `CONDITION`.

Replicated Machines are changed in a more elaborated fashion. First, regarding the alphabet, all CSP paragraphs corresponding to these machines are altered in order to include an argument, `symm`, as their first argument. For instance, the alphabet and the process that correspond to the side door controller would have the following signature:

```
A_S_DOOR_CTRL(symm) = ...
S_DOOR_CTRL(symm) = ...
```

Furthermore, for every variable `v` declared in the **VARIABLES** block of a replicated machine `PP`, we replace any reference to `i_v`, `c_v`, or `o_v`, by `i_v.symm`, `c_v.symm`, or `o_v.symm`, respectively. For example, all references to `c_cond` in the side door controller processes are replaced by `c_cond.symm`.

This concludes the syntactic changes to the result of the basic translation that are needed in presence of replicated machines.

4.2 Synchronized Inputs and Execution Frames

When an input is used in different execution frames, simply synchronising the corresponding machines would deadlock. Our solution is to extend the type of the corresponding channels with the processes IDs. Our example contains two such inputs (see Figure 5): *i_sensor* and *i_slow*. For example, the declaration of channel `i_sensor` is:

```
channel i_sensor : PROCESSES_ID.SIDE_DOOR_CONTROLLER_symm.Bool
```

This changes is reflected to all references to these channels.

To guarantee that the values read in these channels are the same within the same cycle, we create one controller process for each of these channels. It stores the first value read and guarantees that further reads have the same value. The definitions are omitted for conciseness. The final system is composed in parallel with these controllers (defined in process `IN_SYNC_CTRL`) as discussed next.

4.3 Translating the Main System

We assume that the main B machine also follows a pattern, presented below, where inp_i and out_j denote the system's external inputs and outputs.

```
MACHINE $S$ REFINES $S_a$
SEES $p_0.M_0, \ldots, p_i.M_i$
INCLUDES $C_0, \ldots, C_j$
INVARIANT $type(inp_0, \ldots, out_0, \ldots) \wedge safety(inp_0, \ldots, out_0, \ldots)$
OPERATIONS
  $updt(i\_inp_0, ..., i\_inp_n) =$   PRE $type(i\_inp_0) \wedge \ldots \wedge env\_ass(i\_inp_0, ...)$
                                      THEN $F_0; \ldots; F_n$
END
```

Each of the F_i of the update operation corresponds to a frame in which some machines are updated in parallel: $F_i \equiv updt\_C_x(i\_x, \ldots) \, || \ldots || \, updt\_C_y(i\_y, \ldots)$.

The main system might refine an abstract machine. It sees declaration machines such as *CONTEXT* and includes its components. The invariant states type and safety conditions on the inputs and outputs. The main machine has a single update operation, whose precondition specifies the types of the inputs and the assumptions about the environment. Finally, its execution simply updates each component machine using their corresponding inputs as arguments.

```
REFINEMENT $GDC\_r$ REFINES $GDC$
SEES $CONTEXT, ...$
INCLUDES $cmd.CMODE\_DEF, ..., Left.S\_DOOR\_CTRL, Right.S\_DOOR\_CTRL$
OPERATIONS
  $updt\_GDC(i\_ldr, i\_opp, i\_man, i\_ctrl, i\_open\_Left, \ldots, i\_open\_Right, \ldots) =$
  PRE $i\_man \in BOOL \land \ldots$ THEN
      BEGIN $cmd.updt\_CMODE\_DEF(i\_man, i\_ctrl) \parallel \ldots$ END ;
      $cc.updt\_COND(cmd.c\_mode, ld.c\_cabin);$
      BEGIN $Left.updt\_S\_DOOR\_CTRL(i\_open\_Left, \ldots, c\_cond\_Left) \parallel \ldots$ END
  END
END
```

Fig. 5. Sketch of the refined GDC

The pattern is slightly different if we have replicated machines. These changes are very similar to those in the pattern of the component machines. It takes into account that, when instantiating machines, their variable names are changed by prefixing them with the name used in the instantiation.

In Figure 5, we present parts of the definition of the *GDC_r* machine, which refines the abstract *GDC* machine. In order to have access to the system types, the *GDC SEES* the *CONTEXT*. Next, it *INCLUDES* all the system components: the machine *CMODE_DEF* discussed in this paper is one of them. Furthermore, the *GDC* creates two replicated instances of the side door controller (*S_DOOR_CTRL*), named *Left* and *Right*. The *GDC* update simply receives all the inputs and executes each frame in sequence. Each frame is the parallel composition of the update operations of the frame components.

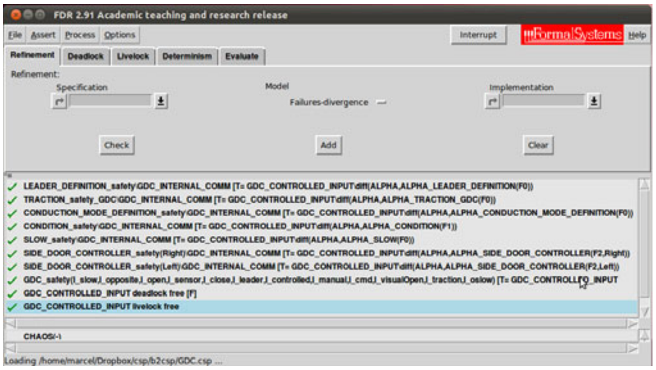
The CSP for the main module declares one integer constant for the `CONTROL` and for each component in the system. Multiple instances of replicated machines are considered as different components.

```
ID_CONTROL                        = 0
ID_CMODE_DEF                      = 1
...
ID_SIDE_DOOR_CONTROLLER(Left)     = 7
MIN_ID                            = 0
MAX_ID                            = length(PROCS)-1
```

The maximum and minimum identification values are used to access two arrays that store the alphabets (`ALPHAS`) and processes (`PROCS`). They are related to the processes identification described in Section 4.2.

```
ALPHAS = < A_CONTROL, A_CMODE_DEF, ..., A_S_DOOR_CTRL(Right)>
PROCS = < CONTROL, CMODE_DEF, ..., S_DOOR_CTRL(Right)>
```

We define the process corresponding to the main system as an indexed parallel composition; each component has its alphabet as interface. The internal



```
Refinement:
  Specification          Model                     Implementation
                         Failures-divergence

✓ LEADER_DEFINITION_safety\GDC_INTERNAL_COMM [T= GDC_CONTROLLED_INPUT\diff(ALPHA,ALPHA_LEADER_DEFINITION(F0))
✓ TRACTION_safety_GDC\GDC_INTERNAL_COMM [T= GDC_CONTROLLED_INPUT\diff(ALPHA,ALPHA_TRACTION_GDC(F0))
✓ CONDUCTION_MODE_DEFINITION_safety\GDC_INTERNAL_COMM [T= GDC_CONTROLLED_INPUT\diff(ALPHA,ALPHA_CONDUCTION_MODE_DEFINITION(F0))
✓ CONDITION_safety\GDC_INTERNAL_COMM [T= GDC_CONTROLLED_INPUT\diff(ALPHA,ALPHA_CONDITION(F1))
✓ SLOW_safety\GDC_INTERNAL_COMM [T= GDC_CONTROLLED_INPUT\diff(ALPHA,ALPHA_SLOW(F0))
✓ SIDE_DOOR_CONTROLLER_safety(Right)\GDC_INTERNAL_COMM [T= GDC_CONTROLLED_INPUT\diff(ALPHA,ALPHA_SIDE_DOOR_CONTROLLER(F2,Right))
✓ SIDE_DOOR_CONTROLLER_safety(Left)\GDC_INTERNAL_COMM [T= GDC_CONTROLLED_INPUT\diff(ALPHA,ALPHA_SIDE_DOOR_CONTROLLER(F2,Left))
✓ GDC_safety(i_slow,i_opposite,i_open,i_sensor,i_close,i_leader,i_controlled,i_manual,i_cmd,i_visualOpen,i_traction,i_oslow) [T= GDC_CONTROLLED_INPUT
✓ GDC_CONTROLLED_INPUT deadlock free [F]
✓ GDC_CONTROLLED_INPUT livelock free

CHAOS(-)
Loading /home/marcel/Dropbox/csp/b2csp/GDC.csp ...
```

Fig. 6. Verification in FDR

communications are hidden from the environment. The function `get(i,S)` returns the `i`-th element of an array `S`.

```
GDC_I = (|| i:{MIN_ID..MAX_ID} @ [get(i,ALPHAS)] get(i,PROCS))
GDC_I_COMM = {| c_mode, c_cabin, c_cond |}
GDC = GDC_I \ GDC_I_COMM
```

Finally, the system is composed with the inputs synchronisation controller previously described. The control signals are also hidden from the environment.

```
GDC_CTRL_INPUT = (GDC [| A_IN_SYNC_CTRL |] IN_SYNC_CTRL) \ A_CTRL_ALL
```

The safety conditions of the main system are translated into a process that offers a choice of inputs and outputs. For replicated machines, the syntactic changes previously described are also applied to the safety process and the transition functions. That includes a new argument, `symm`, as their first argument, and replaces references to `i_v`, `c_v`, or `o_v`, by `i_v.symm`, `c_v.symm`, or `o_v.symm`, respectively. In our example, we get:

```
GDC_safe_Symm(symm, mem, man, ctrl, ldr, opp, open,
                close, sensor, slow, cmd, visual_open, traction, oslow) =
  i_manual?n_manual -> ... [] ...
    [] ((leader and ((man and (not ctrl))
                     or ((not man) and ctrl))
              and (not slow)) &
           o_cmd.symm!CMD_CLOSE -> GDC_safe_Symm(...)) [] ...
```

We are now in a position to present the verification of such systems.

4.4 Verification

As initially intended, the processes created can be model checked. We can, for instance, verify that the main system satisfies the safety requirements of each component separately. This is a safeguard against possible errors in the translation, since such properties have already been proved in the realm of the B method. In each assertion, we hide the events in the alphabet of the relevant component. For example, the assertions below guarantee that GDC_CTRL_INPUT satisfies the safety requirements of the CMODE_DEF component and of one of the side door controllers.

```
assert CMODE_DEF_safe \ GDC_I_COMM [T= GDC_CTRL_INPUT \ diff(A, A_CMODE_DEF)
assert S_DOOR_CTRL_safe(Right) \ GDC_I_COMM [T=
       GDC_CTRL_INPUT \ diff(A, A_S_DOOR_CTRL(Right))
...
```

Finally, we assert that it satisfies its own safety requirements and that it is deadlock-free and livelock-free.

```
assert GDC_safe(...) [T= GDC_CTRL_INPUT
assert GDC_CTRL_INPUT :[ deadlock free [F] ]
assert GDC_CTRL_INPUT :[ livelock free ]
```

The case study generates 830 lines of CSP code containing over 40 processes. Its verification in FDR takes less than 30 seconds; all assertions held in FDR as presented in Figure 6.

5 Conclusions

We presented a mapping from a subset of the B notation to CSP. Such translation allows checking of safety requirements both at the source and the result, thus providing an *a posteriori* verification of the translation. A verification of properties inherent to concurrent systems such as freedom of deadlock and livelock is possible using CSP tool support.

We have intentionally inserted errors in the B specification. Some are not identified using B tools, but have caused a deadlock in the corresponding CSP specification. For instance, we have removed the assignment *mode* in the update operation of the *CMODE_DEF* machine (Figure 4). In the B model, this leaves the value of the variable unchanged and does not harm safety properties nor yields a deadlock. However, the CSP model deadlocks: the process for the *CONDITION* machine waits for an input on *c_mode* that never happens since the B variable is not updated. By tracing back from the CSP model to the B model, it is simple to identify the cause of the deadlock and to fix the problem.

The translation lays the basis for a model-based approach to co-design. Given a B model within the pattern, our methodology yields systems with modules that may be implemented in either software from B, or in hardware from CSP.

There are, however, some issues that still need to be addressed. First, we need to prove the translation correctness. For the moment, we rely on model checking to justify that the properties proved on the B models also hold on the generated CSP processes. In the future, we intend to use the *Unifying Theories of Programming* [9] as a unifying framework of both languages. The semantics of CSP in the UTP has already been presented in [9]; a formal semantics for B in the UTP is the next step.

It is not our goal a translation from any B model to a corresponding CSP model. However, we will extend the pattern to accommodate further models.

Finally, we have implemented a translator that supports the translation presented in this paper and a translator from CSP to Handel-C [13]. In [15], Schneider *et al.* present such translator from CSP||B. Further work is needed to establish a comparison between our approach and theirs.

The full automation of our methodology requires the extension of the CSP features that can be translated into Handel-C [13]: indexed operators, multi-way synchronisation, and hiding are in this extension. All these extensions are currently under development.

References

1. Abrial, J.-R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Butler, M.J.: An approach to the design of distributed systems with B AMN. In: Till, D., Bowen, J., Hinchey, M.G. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 223–241. Springer, Heidelberg (1997)
3. Butler, M.J.: csp2B: a practical approach to combining CSP and B. FACJ 12(3), 182–198 (2000)
4. Déharbe, D., Moreira, A.M., Muniz Silva, P., Russo Jr., A.: Modelling control systems in b: an industrial case study. In: SBMF 2007, pp. 112–127 (2007)
5. Evans, N., Treharne, H.: Linking Semantic Models to Support CSP||B Consistency Checking. Electr. Notes Theor. Comput. Sci. 145, 201–217 (2006)
6. Fischer, C.: CSP-OZ: A combination of Object-Z and CSP. In: FMOODS, vol. 2, pp. 423–438. Chapman & Hall, Boca Raton (1997)
7. Formal Systems Ltd. FDR: User Manual and Tutorial, version 2.82 (2005)
8. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
9. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice-Hall (1998)
10. Mens, T., Van Gorp, P.: A taxonomy of model transformation. ENTCS 152, 125–142 (2006)
11. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
12. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: A UTP Semantics for Circus. FACJ (2008)
13. Oliveira, M.V.M., Woodcock, J.C.P.: Automatic Generation of Verified Concurrent Hardware. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 286–306. Springer, Heidelberg (2007)
14. Roscoe, A.W., Woodcock, J.C.P., Wulf, L.: Non-interference through Determinism. In: Gollmann, D. (ed.) ESORICS 1994. LNCS, vol. 875, pp. 33–54. Springer, Heidelberg (1994)
15. Schneider, S., Treharne, H., McEwan, A., Ifill, W.: Experiments in Translating CSP||B to Handel-C. In: CPA - Communicating Process Architectures Conference. Concurrent Systems Engineering Series, vol. 66, pp. 115–133. IOS Press (2008)
16. Taguchi, K., Araki, K.: The state-based CCS semantics for concurrent Z specification. In: ICFEM, pp. 283–292. IEEE (1997)
17. Woodcock, J.C.P., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall, Englewood Cliffs (1996)

Simulation and Verification of Synchronous Set Relations in Rewriting Logic

Camilo Rocha[1] and César Muñoz[2]

[1] University of Illinois at Urbana-Champaign
[2] NASA Langley Research Center

Abstract. This paper presents a mathematical foundation and a rewriting logic infrastructure for the execution and property verification of synchronous set relations. The mathematical foundation is given in the language of abstract set relations. The infrastructure consists of an order-sorted rewrite theory in Maude, a rewriting logic system, that enables the synchronous execution of a set relation provided by the user. By using the infrastructure, existing algorithm verification techniques already available in Maude for traditional *asynchronous* rewriting, such as reachability analysis and model checking, are automatically available to synchronous set rewriting. The use of the infrastructure is illustrated with an executable operational semantics of a simple synchronous language and the verification of temporal properties of a synchronous system.

1 Introduction

Synchronous set relations provide a natural model for describing the operational semantics of synchronous languages. Previous work by the authors [11] gives a *serialization procedure* for simulating the execution of synchronous set relations by *asynchronous* term rewriting. The synchronous execution of a set relation is a parallel reduction, where the terms to be reduced in parallel are selected according to some strategy. The serialization procedure has been used to provide the rewriting logic semantics of the Plan Execution Interchange Language (PLEXIL) [5], a synchronous plan execution language developed by NASA to support spacecraft automation [6].

Despite being generic, the serialization procedure proposed in [11] has to be coded by the user for each synchronous language. This paper extends that work in two ways. First, it generalizes the theoretical development of synchronous set relations by extending the notion of strategy to enable a larger set of synchronous transformations. Second, it introduces an infrastructure in Maude [4], a high-performance reflective language and system supporting asynchronous set rewriting, that implements on-the-fly a serialization procedure for a synchronous language provided by the user. These contributions allow for simpler and more succinct language specifications, and more general synchronous set relations.

Formally, a synchronous set relation is defined as the *synchronous closure* of an *atomic* relation with a given *strategy*. Two sets are synchronously related if the first set can be transformed into the second set by parallel atomic transformations. The selection of the redexes in the source set is done by the strategy. Strategies can be defined using priorities, which solve conflicts arising from the overlapping of atomic transitions. Section 2 presents, in an abstract setting, definitions of synchronous set relations, strategies, and priorities.

The infrastructure presented in this paper uses the reflection capabilities of Maude's rewriting logic, which is succinctly described in Section 3.1. Maude supports set rewriting, i.e., rewriting modulo axioms such as associativity, commutativity, and identity. These features are well-suited for object-based concurrent systems. The infrastructure consists of a rewrite theory in Maude, defining a set of generic sorts and terms, the algebraic properties of the datatypes, and a set of functions and rewrite rules that support the synchronous execution of an atomic set relation. The infrastructure is described in sections 3.2 and 3.3.

As a direct advantage of using this infrastructure, all commands in Maude for rewrite theories such as its rewrite and search commands, and formal verification tools such as Maude's LTL Model Checker, are available for analyzing properties of synchronous set relations. Section 4 illustrates the use of the infrastructure by giving an executable semantics of a simple synchronous language with arithmetic expressions. Section 5 illustrates the use of Maude's LTL Model Checker for the verification of temporal properties of a synchronous set relation.

The infrastructure in Maude and the examples presented in this paper are available from `http://shemesh.larc.nasa.gov/people/cam/PLEXIL`.

2 Abstract Synchronous Set Relations

This section introduces the concepts of abstract set relations used in this paper.

Let $\mathcal{U}$ be a set whose elements are denoted $A, B, \ldots$ and let $\to$ be a binary relation on $\mathcal{U}$. An element $A \in \mathcal{U}$ is called a $\to$-*redex* if there exists $B \in \mathcal{U}$ such that the *pair* $\langle A\,;B\rangle \in \to$. The expressions $A \to B$ and $A \not\to B$ denote $\langle A\,;B\rangle \in\to$ and $\langle A\,;B\rangle \notin\to$, respectively. The *identity* relation and *reflexive-transitive closure* of $\to$ are defined as usual and denoted $\to^0$ and $\to^*$, respectively.

Henceforth, it is assumed that $\mathcal{U}$ is the family of all *nonempty* finite sets over an abstract and possibly infinite set T, i.e., $\mathcal{U} \subseteq \wp(T)$ and $\emptyset \notin \mathcal{U}$, and, therefore, $\to$ is a binary relation on finite sets of T. The elements of T will be denoted by lowercase letters $a, b, \ldots$. When it is clear from the context, curly brackets are omitted from set notation, e.g., $a, b \to b$ denotes $\{a, b\} \to \{b\}$. Because of this abuse of notation, the symbol ',' is overloaded to denote set union, e.g., if A denotes the set $\{a, b\}$, B denotes the set $\{c, d\}$, and D denotes the set $\{d, e\}$, notation $A, B \to B, D$ denotes $\{a, b, c, d\} \to \{c, d, e\}$.

The *parallel* relation $\to^{\parallel}$ of $\to$ is the relation defined as the parallel closure of $\to$, i.e., the set of pairs $\langle A\,;B\rangle$ in $\mathcal{U} \times \mathcal{U}$ such that $A \to^{\parallel} B$ if and only if there exist $A_1, \ldots, A_n$, (nonempty) pairwise disjoint subsets of A, and sets $B_1, \ldots, B_n$ such that $A_i \to B_i$ and $B = (A \setminus \bigcup_{1 \le i \le n} A_i) \cup \bigcup_{1 \le i \le n} B_i$.

This paper focuses on synchronous set relations. The synchronous relation of an abstract set relation $\to$ is defined as a subset of the parallel closure of $\to$, where a given strategy selects elements from $\to$. Formally, a *$\to$-strategy* is a function s that maps an element $A \in \mathcal{U}$ into a set $s(A) \subseteq \wp(\to)$ such that if $\{\langle A_1 ; B_1\rangle, \ldots, \langle A_n ; B_n\rangle\} \in s(A)$, then $A_i \subseteq A$ and $A_i \to B_i$, for $1 \leq i \leq n$, and $A_1, \ldots, A_n$ are pairwise disjoint.

Definition 1 (Synchronous Relation). *Let s be a $\to$-strategy. The relation $\to^s$ denotes the set of pairs $\langle A ; B\rangle$ in $\mathcal{U} \times \mathcal{U}$ such that $A \to^s B$ if and only if $B = (A \setminus \bigcup_{1\leq i\leq n} A_n) \cup \bigcup_{1\leq i\leq n} B_n$, where $\{\langle A_1 ; B_1\rangle, \ldots, \langle A_n ; B_n\rangle\} \in s(A)$.*

Example 1. Let T be the set of distinct elements a, b, c, d, e, and the relation $\to = \{r_1, r_2, r_3\}$, where $r_1 = \langle a, b ; b, d\rangle$, $r_2 = \langle c ; d\rangle$, and $r_3 = \langle a, c ; e\rangle$. Let s_1, s_2, and s_3 be $\to$-strategies defined for $A = \{a, b, c, d\}$ as follows.

$$s_1(A) = \{\,\{r_2\}, \{r_3\}\,\}, \quad s_2(A) = \{\,\{r_1, r_2\}\,\}, \quad s_3(A) = \{\,\{r_1, r_2\}, \{r_3\}\,\}.$$

It holds that:

$$\begin{array}{lll} a, b, c, d \to^{s_1} a, b, d, & a, b, c, d \to^{s_1} b, d, e, & a, b, c, d \to^{s_2} b, d, \\ a, b, c, d \to^{s_3} b, d, & a, b, c, d \to^{s_3} b, d, e. & \end{array}$$

Some strategies relevant to the operational semantics of synchronous languages are those strategies defined based on a priority. A *priority* $\prec$ for a relation $\to$ is a $\mathcal{U}$-indexed set $\prec = \{\prec_A\}_{A\in\mathcal{U}}$ with each $\prec_A$ a strict partial order on $\to \cap(\wp(A) \times \mathcal{U})$. Priorities can be used to decide between overlapping redexes.

Definition 2 (Saturation). *A set $\{\langle A_1; B_1\rangle, \ldots, \langle A_n; B_n\rangle\} \subseteq \to$ is $\prec$-*saturated *for $A \in \mathcal{U}$ (or $\prec_A$-*saturated*), with $\prec$ be a priority for $\to$, if and only if*

1. *the sets $A_1, \ldots, A_n$ are nonempty pairwise disjoint subsets of A,*
2. *each $\langle A_i ; B_i\rangle$ is such that for any $A' \to B'$ with $A' \subseteq A$ and $A' \cap A_i \neq \emptyset$, $\langle A_i ; B_i\rangle \not\prec_A \langle A' ; B'\rangle$, and*
3. *if there is $A' \to B'$ with $\langle A' ; B'\rangle \notin \{\langle A_1 ; B_1\rangle, \ldots, \langle A_n ; B_n\rangle\}$ and $A' \subseteq A$, then either*
   (i) there is $\langle A_j ; B_j\rangle$, for some $1 \leq j \leq n$, such that $A_j \cap A' \neq \emptyset$ or
   (ii) there is $A'' \to B''$ with $A'' \subseteq A$, $A'' \cap A' \neq \emptyset$, and $\langle A' ; B'\rangle \prec_A \langle A'' ; B''\rangle$.

A $\prec_A$-saturated set is a complete collection of non-overlapping redexes in a term $A \in \mathcal{U}$, where any overlapping is resolved by keeping $\prec$-maximal redexes. Note that the $\prec$-maximality tests in conditions (2) and (3) of Definition 2, are given with respect to *all* pairs $\langle A' ; B'\rangle$ in $\prec_A$, and hence $\prec_A$-saturation exclusively depends on the ordering of the finitely many subsets of $\to \cap(\wp(A) \times \mathcal{U})$.

Example 2. Recall the relation $\to = \{r_1, r_2, r_3\}$ and the set $A = \{a, b, c, d\}$ from Example 1. Let $\prec^1_A$ be such that $r_1 \prec^1_A r_3$. It holds that the sets $\{r_2\}$ and $\{r_3\}$ are $\prec^1_A$-saturated. However, the set $\{r_1, r_2\}$ is not $\prec^1_A$-saturated because r_1 falsifies condition (2) in Definition 2 with witness r_3. Let $\prec^2_A$ be such that $r_3 \prec^2_A r_1$. In this case, the only $\prec^2_A$-saturated set is $\{r_1, r_2\}$. The set $\{r_3\}$ is not $\prec^2_A$-saturated because r_3 falsifies condition (2) in Definition 2 with witness r_1. For $\prec^3_A = \emptyset$, the sets $\{r_1, r_2\}$ and $\{r_3\}$ are the only $\prec^3_A$-saturated sets.

A maximal strategy defines the most general synchronous behavior of a relation, which is given by *all* saturated sets.

Definition 3 (Maximal Strategies). *Let $\prec$ be a priority for $\to$. A $\to$-strategy s is* $\prec$-maximal for $A\in\mathcal{U}$ *(or* $\prec_A$-maximal*) if and only if $s(A)$ is the collection of all $\prec_A$-saturated sets. A $\to$-strategy is $\prec$-maximal if and only if it is $\prec_A$-maximal for all $A\in\mathcal{U}$.*

Example 3. From examples 1 and 2, $\to$-strategies s_1, s_2, and s_3 are, respectively, $\prec^1_A$-maximal, $\prec^2_A$-maximal, and $\prec^3_A$-maximal.

Algorithm 1 witnesses the existence of maximal strategies, which are unique for a given relation $\to$ and a priority $\prec$ (for $\to$).

Theorem 1. *Let $\prec$ be a priority for $\to$. Then a $\prec$-maximal $\to$-strategy exists. Therefore, from Definition 3, the $\prec$-maximal $\to$-strategy is unique.*

Proof. It is proved that the existence of a $\prec$-maximal $\to$-strategy is witnessed by Algorithm 1, for any $A\in\mathcal{U}$ and priority $\prec$ for $\to$. First, the following are important and easy to prove remarks about Algorithm 1:

- all three loops (lines 3, 6, and 12) repeat finitely many times and all quantified conditions (lines 7 and 4) require finitely many comparisons because $A\in\mathcal{U}$ has finitely many elements; also the complexity of γ decreases with each iteration of the third loop, i.e., Algorithm 1 terminates,
- $\alpha = \to \cap(\wp(A)\times\mathcal{U})$ is finite and can be computed effectively,
- $\beta=\alpha\setminus\{\langle A';,B'\rangle\in\alpha \mid (\exists\langle A'';B''\rangle\in\alpha)A'\cap A''\neq\emptyset\wedge\langle A';B'\rangle\prec_A\langle A'';B''\rangle\}$, i.e., β is the subset of α in which all conflicting pairs in α that are not maximal elements in $\prec_A$ have been omitted,
- $\sigma\subseteq\wp(\beta)$ is the collection of largest non-conflicting subsets of β, and
- if $C\in\sigma$, then for any nonempty $C'\subseteq(\beta\setminus C)$, $C\cup C'\notin\sigma$.

Let $D=\{\langle A_1;B_1\rangle,\ldots,\langle A_n;B_n\rangle\}$. It is enough to prove, for $A\in\mathcal{U}$ and priority $\prec$ for $\to$, that D is $\prec_A$-saturated if and only if $D\in\sigma$.

($\Longrightarrow$) If D is $\prec_A$-saturated, then $D\subseteq\alpha$ follows by definition. If $D\not\subseteq\beta$, then there is $\langle A_i;B_i\rangle\in D$ satisfying $\langle A_i;B_i\rangle\prec_A\langle A';B'\rangle$ for some $\langle A';B'\rangle\in\alpha$ with $A'\cap A_i\neq\emptyset$. But then, for D, $\langle A_i;B_i\rangle$ violates condition (2) in Definition 2, a contradiction. Hence $D\subseteq\beta$. If $D\notin\sigma$, since $D\subseteq\beta$ and the $A_1,\ldots,A_n$ are pairwise disjoint by assumption, either there is a nonempty set $D'\subseteq\beta\setminus D$ such that $D\cup D'\in\sigma$ or there is nonempty set $D''\subsetneq D$ such that $D''\in\sigma$. If $D\cup D'\in\sigma$ and since D' is nonempty, any pair $\langle A';B'\rangle\in D'$ violates condition (3.ii) in Definition 2, contradicting the $\prec_A$-maximality of D. If $D''\in\sigma$, then for any pair $\langle A'';B''\rangle\in D\setminus D''$ the set $C=D''\cup\{\langle A'';B''\rangle\}$ falsifies the test in line 14 of Algorithm 1 and hence $C\in\sigma$. Since $D''\in\sigma$ and $D''\subsetneq C\in\sigma$, this contradicts the last remark aforementioned. Therefore, as desired, $D\in\sigma$.

```
Input  : $A \in \mathcal{U}$ and priority $\prec$ for $\to$.
Output: $s(A)$, with $s$ the $\prec_A$-maximal $\to$-strategy.
begin
    $\alpha, \beta, \gamma, \sigma \leftarrow \emptyset, \emptyset, \emptyset, \emptyset$;
    for $A_i$ $\to$-redex, $A_i \subseteq A$, and $B_i$ such that $A_i \to B_i$ do
        add $\langle A_i ; B_i \rangle$ to $\alpha$;
    end
    for $\langle A_i ; B_i \rangle \in \alpha$ do
        if $(\forall \langle A' ; B' \rangle \in \alpha)\ (A_i \cap A') \neq \emptyset \implies \langle A_i ; B_i \rangle \not\prec \langle A' ; B' \rangle$ then
            add $\langle A_i ; B_i \rangle$ to $\beta$;
        end
    end
    $\gamma \leftarrow \{\beta\}$;
    while $\gamma \neq \emptyset$ do
        remove $C$ from $\gamma$;
        if $(\exists \langle A_i ; B_i \rangle, \langle A_j ; B_j \rangle \in C)$ with $i \neq j$ and $A_i \cap A_j \neq \emptyset$
        then add $C \setminus \{\langle A_i ; B_i \rangle\}$ and $C \setminus \{\langle A_j ; B_j \rangle\}$ to $\gamma$;
        else add $C$ to $\sigma$;
    end
    return $\sigma$;
end
```

Algorithm 1. The $\prec$-maximal $\to$-strategy

($\Longleftarrow$) If $D \in \sigma \subseteq \wp(\alpha)$, then $A_1, \ldots, A_n$ are pairwise disjoint $\to$-redexes, thus subsets, of A. Thus, condition (1) in Definition 2 is satisfied. For condition (2), since $D \in \sigma$, it follows that $D \subseteq \beta$. Hence, any $\langle A_i ; B_i \rangle \in D$ satisfies condition (2) in Definition 2. For condition (3), assume there is $\langle A' ; B' \rangle \in \alpha$ with $\langle A' ; B' \rangle \notin D$. Then, either $\langle A' ; B' \rangle \in (\beta \setminus D)$ or $\langle A' ; B' \rangle \in (\alpha \setminus \beta)$. If $\langle A' ; B' \rangle \in (\beta \setminus D)$, then $D \cup \{\langle A' ; B' \rangle\} \notin \sigma$, as previously stated. However, $\langle A' ; B' \rangle \in \beta$, so it must be the case that $A' \cap A_i \neq \emptyset$ for some $1 \leq i \leq n$. If $\langle A ; B' \rangle \in (\alpha \setminus \beta)$, then $\langle A' ; B' \rangle \prec_A \langle A'' ; B'' \rangle$ for some $\langle A'' ; B'' \rangle \in \alpha$. In either case, D satisfies condition (3) in Definition 2. Thus, D is $\prec_A$-saturated. □

The definitions of strategy and maximal strategy used in this paper are more general than those in [11, ∮2]. In that paper, the only possible nondeterminism in $\to^s$ arises from $\to$. In the formalization presented in this paper, as illustrated by strategies s_1 and s_3, the synchronous relation $\to^s$ can be nondeterministic even when the relation $\to$ is deterministic.

3 Synchronous Set Relations in Rewriting Logic

This section presents the infrastructure for specifying and executing in Maude a synchronous relation defined from a language $\mathcal{L}$.

3.1 A Brief Overview of Rewriting Logic

An *order-sorted signature* [2] is a triple $\Sigma = (S, \leq, F)$, where $(S, \leq)$ is a finite poset of sorts and F is a finite set of function symbols. Set $X = \{X_s\}_{s \in S}$ is an S-sorted family of disjoint sets of variables with each X_s countably infinite. The set of terms of sort s is denoted by $T_\Sigma(X)_s$ and the set of ground terms of sort s is denoted by $T_{\Sigma,s}$. It is assumed that for each sort s, $T_{\Sigma,s}$ is nonempty. Algebras $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$ denote the respective term algebras. The set of variables of a term t is written $vars(t)$ and is extended to sets of terms in the natural way. A term t is called *ground* if $vars(t) = \emptyset$. A *substitution* θ is a sorted map from a finite subset $dom(\theta) \subseteq X$ to $ran(\theta) \subseteq T_\Sigma(X)$ and extends homomorphically in the natural way. Substitution θ is called *ground* if $ran(\theta)$ is ground. Expression $t\theta$ denotes the application of θ to term t.

A *Σ-equation* is a sentence $t = u$ **if** *cond*, where $t = u$ is a *Σ-equality* with $t, u \in T_\Sigma(X)_s$, for some sort $s \in S$, and the *condition cond* is a finite conjunction of Σ-equalities. An *equational theory* is a pair (Σ, E) with order-sorted signature Σ and finite set of Σ-equations E. For a Σ-equation φ, the judgement $(\Sigma, E) \vdash \varphi$ states that φ can be derived from (Σ, E) by the deduction rules in [8]. In this case, it holds that φ is valid in all models of (Σ, E). An equational theory (Σ, E) induces the congruence relation $=_E$ on $T_\Sigma(X)$ defined for any $t, u \in T_\Sigma(X)$ by $t =_E u$ if and only if $(\Sigma, E) \vdash (\forall X)\, t = u$. The Σ-algebras $\mathcal{T}_{\Sigma/E}(X)$ and $\mathcal{T}_{\Sigma/E}$ denote the quotient algebras induced by $=_E$ over the algebras $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$. The algebra $\mathcal{T}_{\Sigma/E}$ is called the *initial algebra* of (Σ, E).

A *Σ-rule* is a sentence $\flat : t \Rightarrow u$ **if** *cond*, where $\flat$ is its *name*, $t \Rightarrow u$ is a *Σ-sequent* with $t, u \in T_\Sigma(X)_s$, for some sort $s \in S$, and the *condition cond* is a finite conjunction of Σ-equations. A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E, R)$ with equational theory $\mathcal{E}_\mathcal{R} = (\Sigma, E)$ and a finite set of Σ-rules R. For $\mathcal{R} = (\Sigma, E, R)$ and $\flat$ a Σ-rule, the judgement $\mathcal{R} \vdash \flat$ states that $\flat$ can be derived from $\mathcal{R}$ by the deduction rules in [2]. In this case, it holds that $\flat$ is valid in all models of $\mathcal{R}$. For $\flat$ a Σ-equation, it can be proved that $\mathcal{R} \vdash \flat$ if and only if $\mathcal{E}_\mathcal{R} \vdash \flat$. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ induces the rewrite relation $\Rightarrow_\mathcal{R}$ on $T_{\Sigma/E}(X)$ defined for every $t, u \in T_\Sigma(X)$ by $[t]_E \Rightarrow_\mathcal{R} [u]_E$ if and only if there is a *one-step* rewrite proof $\mathcal{R} \vdash (\forall X)\, t \Rightarrow u$. Relations $\Rightarrow_\mathcal{R}$ and $\Rightarrow^*_\mathcal{R}$ respectively denote a one-step rewrite and an arbitrary length (but finite) rewrite in $\mathcal{R}$ from t to u. Model $\mathcal{T}_\mathcal{R} = (\mathcal{T}_{\Sigma/E}, \Rightarrow^*_\mathcal{R})$ is the *initial reachability model* of $\mathcal{R} = (\Sigma, E, R)$ [2].

The following conditions on a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ make rewriting with equations E and with rules R modulo E computable, and are assumed throughout this paper. First the set of equations E of $\mathcal{R}$ can be decomposed into a disjoint union $E' \uplus A$, with A a collection of axioms (such as associativity, and/or commutativity, and/or identity) for which there exists a *matching algorithm modulo A* producing a finite number of A-matching substitutions, or failing otherwise. The second condition is that the equations E' can be oriented into a set of *ground sort-decreasing*, *ground confluent*, and *ground terminating* rules $\overrightarrow{E'}$ modulo A. The expression $[\mathrm{can}_{\Sigma,E'/A}(t)]_A \in T_{\Sigma/A,s}$ will denote the *E'-canonical form* of $[t]_A$. The rules R in $\mathcal{R}$ are assumed to be *ground coherent* relative to the equations E' modulo A [14].

3.2 The Synchronous Language $\mathcal{L}$

Recall that definitions in Section 2 are given for an abstract set T, an abstract relation $\rightarrow$, and an abstract priority relation $\prec$. The language $\mathcal{L}$ is given by the user as an order-sorted rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ that enables the definition of concrete mathematical objects $T_{\Sigma_{\mathcal{L}},Elem}$, $\rightarrow_{\mathcal{L}}$, and $\prec_{\mathcal{L}}$ that implement T, $\rightarrow$, $\prec$, respectively. The rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ extends the rewrite theory (Σ, E, R), which provides an infrastructure with definitions of basic sorts and data structures that are suitable for specifying set rewriting systems. This rewrite theory exploits rewriting logic's reflection capabilities available in Maude to *soundly* and *completely* simulate the synchronous relation $\rightarrow^{s}_{\mathcal{L}}$, where s is the $\prec_{\mathcal{L}}$-maximal strategy for $\rightarrow_{\mathcal{L}}$.

The Set $T_{\Sigma_{\mathcal{L}},Elem}$. The set of ground terms $T_{\Sigma_{\mathcal{L}},Elem}$ of the rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ implements the abstract set T of Section 2. The sort *Elem* represents *elements* in Σ having the form $\langle m \mid a_1 : e_1, \ldots, a_n : e_n\rangle$, where m is an identifier of sort *Eid* and $a_1 : e_1, \ldots, a_n : e_n$ is a *map* of sort *Map*. A map is a collection of *attributes*. An *attribute* is a pair $a : e$ where a is an attribute identifier of sort *Aid* and e is an expression of sort *Expr*. Attributes are a flexible way of defining the internal state of an element. Sorts *Aid* and *Eid* are declared as subsorts of *Expr*. The set $\mathcal{U}$ of Section 2 corresponds to the set of ground terms $T_{\Sigma_{\mathcal{L}},Ctx}$, where the sort *Ctx* represents sets of elements of sort *Elem*. A *context* is an element of sort *Ctx*. The sort *Val* is defined in Σ as a subsort of *Expr* and represents built-in values such as Boolean and numerical values. Function symbol $eval : Ctx \times Expr \longrightarrow Val$ is defined in Σ without any equational definition.

The user is free to extend the signature Σ in $\Sigma_{\mathcal{L}}$ with any syntax and subsorts for element identifiers, attribute identifiers, and expressions. However, it is assumed that attribute identifiers within a map and element identifiers within a context are unique. It is also assumed that the theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$ includes a complete equational interpretation of *eval* for the set of expressions in $\Sigma_{\mathcal{L}}$.

The Relation $\rightarrow_{\mathcal{L}}$. The synchronous relation in Definition 1 is given for an abstract atomic relation $\rightarrow$. In a concrete language, such as $\mathcal{L}$, this relation represents atomic computational steps that are synchronously executed. For that reason, the concrete relation $\rightarrow_{\mathcal{L}}$ is called the *atomic relation*. As shown in [11], the atomic relation is usually parametric with respect to a context that, in this infrastructure, provides global information to the function *eval*. Henceforth, the atomic relation with respect to a context Γ of sort *Ctx* will be denoted $\overset{\Gamma}{\rightarrow}_{\mathcal{L}}$.

The atomic relation $\rightarrow_{\mathcal{L}}$ is specified in $R_{\mathcal{L}}$ through *atomic rules*.

Definition 4 (Atomic Rules). *Let $\Sigma_{\mathcal{L}}$ be an order-sorted signature extending Σ. An* atomic $\Sigma_{\mathcal{L}}$-rule *is a $\Sigma_{\mathcal{L}}$-rule $\flat : l \Rightarrow r$* **if** *cond such that:*

- *rule name $\flat$ has the form c–n, where c, the* component *of $\flat$, is an identifier, and n, the* rank *of $\flat$, is a natural number;*
- *l does not contain attribute identifier variables, i.e., $vars(l) \cap X_{Aid} = \emptyset$; and*

– *attribute names appearing in an element term in r are named for that same element term in l, i.e., if $\langle i \mid m' \rangle \in r$ and $(a : e') \in m'$, then there is $\langle i \mid m \rangle \in l$ such that $(a{:}e) \in m$ for some $e \in T_{\Sigma_{\mathcal{L}}}(X)_{Expr}$.*

An atomic $\Sigma_{\mathcal{L}}$-rule specifies transitions of contexts (possibly) constrained by a condition that may involve expressions in the syntax of $\mathcal{L}$. The component and rank of $\Sigma_{\mathcal{L}}$-rules are used to define the priority relation $\prec_{\mathcal{L}}$. The restriction on attribute identifier names and variables is to prevent the user from defining an atomic relation $\rightarrow_{\mathcal{L}}$ for which computing a $\rightarrow_{\mathcal{L}}$-reduction could be highly inefficient or even incorrect.

Definition 5 (Atomic Relation $\rightarrow_{\mathcal{L}}$). *Let $\mathcal{L} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ be a rewrite theory with $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$ extending (Σ, E) and $R_{\mathcal{L}}$ a collection of atomic $\Sigma_{\mathcal{L}}$-rules with different names. For a rule $\flat : l \Rightarrow r$* **if** *$cond \in R_{\mathcal{L}}$, the (parametric) relation $\overset{\Gamma}{\rightarrow}_{\flat}$, with parameter $\Gamma \in T_{\Sigma_{\mathcal{L}},Ctx}$, denotes the set of pairs $\langle A\,;B\rangle$ in $T_{\Sigma_{\mathcal{L}},Ctx} \times T_{\Sigma_{\mathcal{L}},Ctx}$ such that there is a ground substitution $\theta : T_{\Sigma_{\mathcal{L}}}(X) \longrightarrow T_{\Sigma_{\mathcal{L}}}$ satisfying $cond\theta$, $A = l\theta$, and $B = r\theta$ in $\mathcal{L}$, where any expression is evaluated in Γ. The* atomic relation *$\rightarrow_{\mathcal{L}}$ is the indexed set $\{\overset{\Gamma}{\rightarrow}_{\flat}\}_{\Gamma \in T_{\Sigma_{\mathcal{L}},Ctx}, \flat \in R_{\mathcal{L}}}$.*

In Definition 5, A, B, and Γ are ground terms of sort *Ctx*. Furthermore, the term B is a variant of A in which some expressions and attributes have been modified. In particular, A and B have the same number of elements with the same element and attribute identifiers. This means that the atomic relation does not delete or create elements or attributes in A. This restriction simplifies the technical development of (Σ, E, R). In any case, creation and deletion of elements and attributes can be encoded by using additional attributes. Also observe that, due to the syntactical restrictions of atomic rules in Definition 4, equational sentences $C\theta$, $A = l\theta$, and $B = r\theta$ can be checked in $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$ because they are equational expressions that, although may depend on context Γ, do *not* depend on $R_{\mathcal{L}}$.

In general, the atomic relation $\rightarrow_{\mathcal{L}}$ and the rewrite relation $\Rightarrow_{\mathcal{L}}$ induced by the rewrite theory $\mathcal{L}$ do not coincide for ground context terms. In particular, $\rightarrow_{\mathcal{L}}$ is defined as the top-most application of the atomic rules, while $\Rightarrow_{\mathcal{L}}$ is defined as the congruence closure of those rules.

The Priority $\prec_{\mathcal{L}}$. For a given context Γ, the elements in $\overset{\Gamma}{\rightarrow}_{\mathcal{L}}$ can be regarded as tuples of the form $(A, B, c, m)_{\Gamma}$ as a shorthand for $A \overset{\Gamma}{\rightarrow}_{c\text{–}m} B$, with $c\text{–}m \in R_{\mathcal{L}}$. The set $\prec_{\mathcal{L}} = \{\prec_{\mathcal{L}(\Gamma)}\}_{\Gamma \in T_{\Sigma_{\mathcal{L}},Ctx}}$ is defined automatically by the infrastructure:

$$(A', B', c', m')_{\Gamma} \prec_{\mathcal{L}(\Gamma)} (A, B, c, m)_{\Gamma} \quad \equiv \quad A \subseteq \Gamma \wedge A' \subseteq \Gamma \wedge c = c' \wedge m < m',$$

where $<$ is the usual order on natural numbers.

Lemma 1. *The indexed set $\prec_{\mathcal{L}}$ is a priority for $\rightarrow_{\mathcal{L}}$.*

Proof. It is enough to prove that $\prec_{\mathcal{L}(\Gamma)}$ is a strict partial order, for any $\Gamma \in T_{\Sigma_{\mathcal{L}},Ctx}$. Irreflexivity of $\prec_{\mathcal{L}(\Gamma)}$ follows from the irreflexivity of $<$. Transitivity of $\prec_{\mathcal{L}(\Gamma)}$ follows from the fact that if $(A'', B'', c'', m'')_{\Gamma} \prec_{\mathcal{L}(\Gamma)} (A', B', c', m')_{\Gamma}$ and $(A', B', c', m')_{\Gamma} \prec_{\mathcal{L}(\Gamma)} (A, B, c, m)_{\Gamma}$, then $A'' \subseteq \Gamma$, $A \subseteq \Gamma$, $c'' = c' = c$, and $m < m' < m''$. Therefore, $(A'', B'', c'', m'')_{\Gamma} \prec_{\mathcal{L}(\Gamma)} (A, B, c, m)_{\Gamma}$. □

The priority $\prec_{\mathcal{L}}$ is an indexed collection of strict partial orders. In particular, for each $\Gamma \in T_{\Sigma_{\mathcal{L}},Ctx}$, priority $\prec_{\mathcal{L}(\Gamma)}$ compares two elements of $\rightarrow_{\mathcal{L}}$ if they are computed with the same context and they originate from atomic $\Sigma_{\mathcal{L}}$-rules having the same component. It assigns a higher priority to elements with smaller rank.

Rewrite theory (Σ, E, R) includes a function *max-strat* that computes the $\prec_{\mathcal{L}}$-maximal $\overset{\Gamma}{\rightarrow}_{\mathcal{L}}$-strategy, where $\Gamma \in T_{\Sigma_{\mathcal{L}},Ctx}$ is the parameter of the relation $\rightarrow_{\mathcal{L}}$. That function implements Algorithm 1 of Section 2. It takes as input the language $\mathcal{L}$ and ground context Γ and returns the collection $s(\Gamma)$, where s is the $\prec_{\mathcal{L}}$-maximal $\rightarrow_{\mathcal{L}}$-strategy. The function *max-strat* is implemented in Maude using the meta-level capabilities of the system. Henceforth, the strategy s will denote the $\prec_{\mathcal{L}}$-maximal $\rightarrow_{\mathcal{L}}$-strategy as computed by *max-strat*.

3.3 Simulation of $\rightarrow^{s}_{\mathcal{L}}$

The set of Σ-rules R of the order-sorted rewrite theory (Σ, E, R) includes only one rule: for $l, r \in X_{Ctx}$, $T \in X_{Transition}$, and $S \in X_{TransitionSet}$

$$\begin{aligned} \mathit{sync} : \{l\} \Rightarrow \{r\} \quad \textbf{if} \quad & T, S := \mathit{max\text{-}strat}(\mathcal{L}, l) \\ \wedge \quad & r := \mathit{update}(l, T). \end{aligned}$$

This rule, along with the rules $R_{\mathcal{L}}$ provided by the user, implements the serialization algorithm defined in [11], which has been adapted to the notion of maximal strategy presented in this paper. Sort *Transition* denotes sets of pairs in $T_{\Sigma_{\mathcal{L}}}(X)_{Ctx}$ and sort *TransitionSet* denotes collections of transitions. Function *update* takes as inputs a ground context A and a ground transition term $C = \{\langle A_1 \, ; B_1\rangle, \ldots, \langle A_n \, ; B_n\rangle\}$, and computes the ground context $B = (A \setminus \bigcup_{1\leq i\leq n} A_i) \cup \bigcup_{1\leq i\leq n} B_i$.

It is noted that the rule *sync* acts on contexts that are syntactically wrapped by curly brackets, that is, terms of the form $\{A\}$ with A a ground context term. Those terms are of sort *SState*. The curly brackets operator prevents its context A to be directly rewritten by the user defined atomic rules in $R_{\mathcal{L}}$. The actual application of those rules is done by the function *update*.

Rule *sync* is *nondeterministic* because a ground substitution for l matching its condition depends on the choice of T, i.e., on *all* possible transitions computed by *max-strat*. However, there will be *exactly* one rewrite with *sync* for each transition.

Theorem 2. *Let $\mathcal{L} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ be an extension of (Σ, E, R). For $A, B \in T_{\Sigma_{\mathcal{L}},Ctx}$, the following equivalence holds:*

$$\mathcal{L} \vdash \{A\} \Rightarrow \{B\} \qquad \equiv \qquad A \rightarrow^{s}_{\mathcal{L}} B,$$

where s denotes the $\prec_{\mathcal{L}}$-maximal $\rightarrow_{\mathcal{L}}$-strategy as computed by max-strat.

Proof. The key observation is that because *max-strat* computes the $\prec_{\mathcal{L}}$-maximal $\rightarrow_{\mathcal{L}}$-strategy s, the following equivalence holds:

$$C \in s(A) \qquad \equiv \qquad (\exists C' \in T_{\Sigma_{\mathcal{L}},TransitionSet}) C, C' =_{E_{\mathcal{L}}} \mathit{max\text{-}strat}(\mathcal{L}, A).$$

($\Longrightarrow$) Since $\{A\}$ can be rewritten only by rule *sync* $\in R$, there is a ground substitution $\theta : X \longrightarrow T_{\Sigma_{\mathcal{L}}}$ satisfying $A =_{E_{\mathcal{L}}} l\theta$, $B =_{E_{\mathcal{L}}} r\theta$, $T\theta, S\theta =_{E_{\mathcal{L}}}$ *max-strat*$(\mathcal{L}, l\theta)$, and $r\theta =_{E_{\mathcal{L}}}$ *update*$(l\theta, T\theta)$. By the observation above, $T\theta \in s(A)$. Then, from the definition of *update*, it follows that $A \rightarrow^s_{\mathcal{L}} B$.

($\Longleftarrow$) If $A \rightarrow^s_{\mathcal{L}} B$, there is $C = \{\langle A_1\,; B_1\rangle, \ldots, \langle A_n\,; B_n\rangle\} \in s(A)$ such that $B = (A \setminus \bigcup_{1\leq i\leq n} A_i) \cup \bigcup_{1\leq i\leq n} B_i$. By the observation above and the definition of *update*, there is $C' \in T_{\Sigma_{\mathcal{L}}, TransitionSet}$ such that $C, C' =_{E_{\mathcal{L}}}$ *max-strat*$(\mathcal{L}, A)$ and $B =_{E_{\mathcal{L}}}$ *update*(A, C). Then substitution θ satisfying $A =_{E_{\mathcal{L}}} l\theta$ witnesses $\mathcal{L} \vdash \{A\} \Rightarrow \{B\}$. □

One key advantage of this approach is that, while it offers support for the execution of a synchronous relation $\rightarrow^s_{\mathcal{L}}$, it does that by simulating $\rightarrow^s_{\mathcal{L}}$ using the standard asynchronous semantics of Maude. Therefore, all commands available in Maude for executing and verifying rewrite relations are directly available for $\rightarrow^s_{\mathcal{L}}$. Sections 4 and 5 illustrate these features with practical examples.

4 Executable Semantics of a Simple Synchronous Language

Module *SMAUDE* implements in Maude the rewrite theory (Σ, E, R) presented in Section 3. This section illustrates the use of *SMAUDE* by giving the small-step semantics of a simple synchronous language with arithmetic expressions.

Consider a language that consists of two kinds of elements: *memory* elements $\texttt{Mem}(m, v)$ and *assignment* elements $l{:=}e$, where m, l denote memory names, v denotes a numerical value, and e denotes an arithmetic expression. Arithmetic expressions are recursively formed using memory names, numerical values, and expressions of the form $e_1 + e_2$, where e_1 and e_2 are arithmetic expressions. In this case, set T consists of all elements having the form $\texttt{Mem}(m, v)$ or $m{:=}v$.

The small-step semantics of the language requires the definition of an evaluation function *eval* that takes as inputs a context Γ, which is a set of elements T, and an arithmetic expression e. It is inductively defined on expressions:

$$eval(\Gamma, e) = \begin{cases} v & \text{if } e \text{ is the numerical value } v, \\ v & \text{if } e \text{ is the memory name } m \text{ and } \texttt{Mem}(m, v) \in \Gamma, \\ v_1 + v_2 & \text{if } e \text{ has the form } e_1 + e_2,\ v_i = eval(\Gamma, e_i) \text{ for } i \in \{1, 2\}. \end{cases}$$

The (parametric) atomic relation $\rightarrow$ of the language is defined for a context Γ by $A \xrightarrow{\Gamma} B$ if and only if $A \subseteq \Gamma$, $A = \{\texttt{Mem}(m, v), l{:=}e\}$, $B = \{\texttt{Mem}(m, u), l{:=}e\}$, and $u = eval(A, e)$, for some memory name m, values v and u, and expression e. The semantic relation of the language is the relation $\xrightarrow{\Gamma}^{s}$, where s is the $\prec$-maximal $\xrightarrow{\Gamma}$-strategy, Γ is a ground context, and $\prec$ is the empty priority.

Example 4. Let $\Gamma = \{\texttt{Mem}(x, 3), \texttt{Mem}(y, 4), x{:=}y, y{:=}x\}$. Then:

$$\texttt{Mem}(x, 3), \texttt{Mem}(y, 4), x{:=}y, y{:=}x \xrightarrow{\Gamma}^{s} \texttt{Mem}(x, 4), \texttt{Mem}(y, 3), x{:=}y, y{:=}x.$$

This language is specified by the Maude module *SIMPLE*, which includes system module *SMAUDE*:

$$\begin{array}{lll} a : Nat \to Eid & body : \ \to Aid & Nat \leq Val \\ x : \quad\ \to Eid & mem : \ \to Aid & + : Expr \times Expr \to Expr \\ y : \quad\ \to Eid & to : \ \to Aid & \end{array}$$

Memory elements use constructors x and y for element identifiers and have attribute *mem* as their only attribute. Assignment elements use constructors a for element identifiers and have attributes *body* and *to* as their only attributes. In the syntax of *SIMPLE*, memory element $\texttt{Mem}(x, v)$ and an assignment element $x{:=}e$ are represented, for instance, by elements $\langle x \mid mem : v\rangle$ and $\langle a(1) \mid to : x, body : e\rangle$, respectively. Built-in natural numbers are values of the language. Evaluation of expressions is given equationally following the definition of *eval*.

Atomic rule r–1 specifies the atomic relation of the language:

$$r\text{–}1 : \langle I \mid mem{:}N\rangle\langle J \mid body{:}E, to{:}I\rangle \Rightarrow \langle I \mid mem{:}eval(E)\rangle.$$

The specification of atomic rules is slightly different to the usual specification of rules in rewriting logic. First, in the lefthand side of an atomic rule, it is sufficient to only mention the attributes involved in the atomic transition. In this case, *SMAUDE* will complete each lefthand side term by automatically adding a variable of sort *Map*, unique for each element, before any matching is performed. Second, in the righthand side of an atomic rule, it is sufficient to only mention the elements and the attributes that can change in the atomic step. In this case, *SMAUDE* updates in the current state *only* the attributes of the elements occurring in the righthand side of the rule, while keeping the other ones intact. So, in atomic rule r–1, the only attribute that can change is attribute *mem* of the memory element. Note also that in the righthand side of r–1 a unary version of function *eval*, without mention to any particular context, is used; *SMAUDE* will automatically extend it to its binary counterpart, for the given context, when computing function *max-strat*.

The context Γ in Example 4, written in the syntax of *SIMPLE*, is

$$\langle x \mid mem{:}3\rangle\langle y \mid mem{:}4\rangle\langle a(1) \mid to{:}x, body{:}y\rangle\langle a(2) \mid to{:}y, body{:}x\rangle.$$

Maude's *search* command can be used to compute, for instance, the one-step synchronous semantic relation of the language in Example 4 from context Γ:

```
Maude> search { Gamma } =>1 X:SState .
search in SIMPLE : { Gamma } =>1 X:SState .
Solution 1 (state 1)
states: 2  rewrites: 514 in 53ms cpu (54ms real) (9655 rewrites/second)
X:SState --> {< x | mem : 4 > < y | mem : 3 >
                < a(1) | body : y, to : x > < a(2) | body : x, to : y > }
No more solutions.
```

5 Verification of Synchronous Relations

This section illustrates the use of Maude's LTL Model Checker for the verification of properties of a synchronous relation.

Consider a system of *clocks* keeping track of hours and minutes. Each clock is modeled in rewrite theory *CLOCKS* by two elements, one displaying hours and the other displaying minutes:

$$
\begin{array}{lll}
h : \mathit{Nat} \to \mathit{Eid} & \mathit{hour} : \ \to \mathit{Aid} & \mathit{min} \ : \ \mathit{Nat} \to \mathit{Expr} \\
m : \mathit{Nat} \to \mathit{Eid} & \mathit{min} : \ \to \mathit{Aid} & \mathit{Nat} \leq \mathit{Val}
\end{array}
$$

Hour elements use constructor h for element identifiers and have attribute *hour* as its single attribute. Minute elements use constructor m for element identifiers and have attribute *min* as its single attribute. Natural numbers are used as values for the attributes. The n-th clock is represented by the hour element with element identifier $h(n)$ and the minute element with element identifier $m(n)$. Attribute *min* of a minute element $m(n)$ can be accessed by evaluating expression *min(n)*. A clock displaying 9:15, written in the syntax of *CLOCKS*, is

$$\langle h(1) \mid \mathit{hour}\!:\!9\rangle \ \langle m(1) \mid \mathit{min}\!:\!15\rangle .$$

The following clock transitions are of interest:

(i) if $\mathit{hour}{=}11$ and $\mathit{min}{=}59$, then set $\mathit{hour} = 0$ and $\mathit{min} = 0$;
(ii) if $\mathit{hour}{<}11$ and $\mathit{min}{=}59$, then increment *hour* in one unit and set $\mathit{min} = 0$;
(iii) if $\mathit{hour}{<}11$ and $\mathit{min}{<}59$, then increment *min* in one unit.

These transitions are intuitively coded via priorities in rewrite theory *CLOCKS*. The behavior of the system is modeled by defining a priority such that redexes of the form (i) have the highest priority and the ones of the form (iii) the lowest. The following are the atomic rules of *CLOCKS*, for $C, M, N \in X_{\mathit{Nat}}$:

$$
\begin{array}{lll}
\mathit{cl}\text{–}1 \ : & \langle h(C) \mid \mathit{hour}\!:\!11\rangle & \\
 & \langle m(C) \mid \mathit{min}\!:\!59\rangle \ \Rightarrow & \langle h(C) \mid \mathit{hour}\!:\!0\rangle\langle m(C) \mid \mathit{min}\!:\!0\rangle \\
 & & \\
\mathit{cl}\text{–}2 \ : & \langle h(C) \mid \mathit{hour}\!:\!N\rangle \ \Rightarrow & \mathit{if} \ \mathit{eval}(\mathit{min}(C)) == 59 \\
 & & \mathit{then} \ \langle h(C) \mid \mathit{hour}\!:\!s(N)\rangle \\
 & & \mathit{else} \ \langle h(C) \mid \mathit{hour}\!:\!N\rangle \ \mathit{fi} \\
 & & \\
\mathit{cl}\text{–}3 \ : & \langle m(C) \mid \mathit{min}\!:\!N\rangle \ \Rightarrow & \mathit{if} \ N == 59 \\
 & & \mathit{then} \ \langle m(C) \mid \mathit{min}\!:\!0\rangle \\
 & & \mathit{else} \ \langle m(C) \mid \mathit{min}\!:\!s(N)\rangle \ \mathit{fi}
\end{array}
$$

In *CLOCKS*, *resetting* a clock (i.e., rule *cl*–1) has higher priority than exclusively increasing the hour (i.e., rule *cl*–2) or the minute (i.e., rule *cl*–3) of a clock. Rule *cl*–1 uses matching for detecting when a clock needs to be reset. In the righthand side of rule *cl*–2 the evaluation of expression $\mathit{min}(C)$ will yield the minute value of clock C, freeing the lefthand side of the rule from explicitly mentioning the minutes element. Because of this, rules *cl*–2 and *cl*–3 can never overlap and therefore can be executed in parallel. As a final remark, observe that the priorities of the last two rules can be switched without altering the behavior of the system, since their lefthand sides can never overlap.

Two temporal properties that the synchronous relation of *CLOCKS* must satisfy is that clocks are *always synchronized* and that each clock is *reset* infinitely often. These two properties are specified by propositions $\Pi = \{sync, reset\}$. Using the syntax of Maude's LTL Model Checker and for variables $C, C', H, M \in X_{Nat}$ and $\Gamma \in X_{Ctx}$, the propositions Π are defined in the equational theory *CLOCKS-PREDS* as follows:

$$sync : Nat \times Nat \to Prop \qquad reset : Nat \to Prop \qquad SState \leq State$$

$$\{\Gamma\} \models sync(C, C') = \begin{cases} true & \text{if } \langle h(C) \mid hour\!:\!H\rangle\langle m(C) \mid min\!:\!M\rangle \subseteq \Gamma \\ & \wedge \langle h(C') \mid hour\!:\!H\rangle\langle m(C') \mid min\!:\!M\rangle \subseteq \Gamma, \\ false & \text{otherwise.} \end{cases}$$

$$\{\Gamma\} \models reset(C) = \begin{cases} true & \text{if } \langle h(C) \mid hour\!:\!0\rangle\langle m(C) \mid min\!:\!0\rangle \subseteq \Gamma, \\ false & \text{otherwise.} \end{cases}$$

The subsort declaration $SState \leq State$ tells Maude's LTL Model Checker that the semantics of propositions Π (each with sort *Prop* –provided by the model checker) is to be defined on sort *SState*. Two clocks are synchronized if their hour values and minute values are the same; otherwise they are not synchronized. A clock is reset if its hour and minute values are 0.

Consider the following state *init* in the signature of *CLOCKS*

$$\{\langle h(1) \mid hour\!:\!0\rangle\langle m(1) \mid min\!:\!0\rangle\langle h(2) \mid hour\!:\!0\rangle\langle m(2) \mid min\!:\!0\rangle\},$$

with two clocks, both displaying 0:00. The two temporal properties aforementioned that the synchronous relation of *CLOCKS* must satisfy, are formally specified for state *init* as follows:

$$\mathcal{K}^{\Pi}_{CLOCKS}, init \models \Box sync(1, 2),$$
$$\mathcal{K}^{\Pi}_{CLOCKS}, init \models \Box\Diamond reset(1) \ \wedge\ \Box\Diamond reset(2),$$

where $\mathcal{K}^{\Pi}_{CLOCKS} = (T_{\Sigma/E,SState}, \Rightarrow_{CLOCKS}, L_{\Pi})$ is the Kripke structure associated to the initial reachability model $\mathcal{T}_{CLOCKS}$, with topsort *SState*, and predicates Π (see [4] for details on how $\mathcal{K}^{\Pi}_{CLOCKS}$ is associated to $\mathcal{T}_{CLOCKS}$).

First observe that the set of clock states reachable from *init* is finite and, therefore, each property specification problem is decidable. The first property specification asserts that clocks 1 and 2 are always synchronized, and the second property specification asserts that each clock is reset infinitely often.

By using Maude's LTL Model Checker, the following results are obtained:

```
Maude> red modelCheck(init, [] sync(1,2)) .
reduce in CLOCKS-PREDS : modelCheck(init, []sync(1, 2)) .
rewrites: 124946 in 6023ms cpu (6023ms real) (20744 rewrites/second)
result Bool: true

Maude> red modelCheck(init, ([] <> reset(1)) /\ ([] <> reset(2))) .
reduce in CLOCKS-PREDS : modelCheck(init, []<> reset(1) /\ []<> reset(2)) .
rewrites: 125514 in 6810ms cpu (6812ms real) (18428 rewrites/second)
result Bool: true
```

6 Conclusion

Rewriting logic has been used previously as a test bed for specifying and animating synchronous rewrite relations. M. AlTurki and J. Meseguer [1] have studied the rewriting logic semantics of the language Orc, which includes a synchronous reduction relation. T. Serbanuta *et al.* [13] and C. Chira *et al.* [3] define the execution of P-systems with structured data with continuations. The focus of the former is to use rewriting logic to study the (mainly) non-deterministic behavior of Orc programs, while the focus of the latter is to study the relationship between P-systems and the existing continuation framework for enriching each with the strong features of the other. D. Lucanu [7] studies the problem of the interleaving semantics of concurrency in rewriting logic for synchronous systems from the perspective of P-systems. More recently, T. Serbanuta [12] advances the rewriting-based framework $\mathbb{K}$ with resource sharing semantics that enables some kind of synchronous rewriting. J. Meseguer and P. Ölveczky [9] present a formal specification of the *physically asynchronous logically synchronous* architectural pattern as a formal model transformation that maps a synchronous design, together with performance bounds on the underlying infrastructure, to a formal distributed real-time specification that is semantically equivalent to the synchronous design.

The work presented in this paper is closely related to those works in that it presents techniques for specifying and executing synchronous rewrite relations. However, the work presented here is a first milestone towards the development of *symbolic* techniques for the analysis of synchronous set relations. In particular, the authors strongly believe that the infrastructure presented in Section 3 can be extended with rewriting and narrowing based techniques, in the style of [10], to obtain a deductive approach for verifying symbolic safety properties, such as invariance or race conditions, of synchronous set relations. Another feature that distinguishes this work from related work is the idea of priorities as an instrument to control nondeterminism of synchronous relations. Of course, in some cases priorities can be encoded in the condition of rewrite rules, but the treatment here seems more convenient and simpler for the end-user. One interesting exercise would be to study how best to implement this feature in the framework $\mathbb{K}$ and for real-time specifications in rewriting logic.

The contribution of this paper to rewriting logic research is the implementation of general synchronous set relations via asynchronous set rewrite systems. This work extends previous work reported in [11] by giving an on-the-fly implementation of the serialization procedure for rewrite theories that supports execution and verification of more general synchronous set relations. The framework exploits rewriting logic's reflective capabilities, and its implementation in Maude, to soundly and completely simulate the synchronous relation associated to an atomic relation and a maximal strategy specified by atomic rules. This work also generalizes the concept of priority, so that more general synchronous set relations are supported both theoretically and in the Maude infrastructure. A priority, as treated in this work, allows for nondeterministic synchronous relations even when the atomic relation is deterministic. In [11], the only possible nondeterminism in a synchronous relations arises from its atomic relation. A direct benefit to the user from using the infrastructure presented in this paper, is the wealth of Maude's *ground* analysis tools for rewrite theories such as its rewrite and search commands, and its LTL Model Checker.

Although the framework is illustrated with simple examples, it is currently being used to specify an executable semantics in Maude of the Plan Execution Interchange Language (PLEXIL) [5], an open source synchronous language developed by NASA to support autonomous spacecraft operations. This specification enables the application of formal verification techniques available in Maude, such as model-checking and reachability analysis, to PLEXIL programs.

The Maude infrastructure presented in this work is a first prototype of the theoretical developments. Future work includes the development of a wider range of case studies stressing the infrastructure's capabilities; it is also important to streamline the algorithms and data structures in the infrastructure. Future work in the area of deductive analysis will study symbolic reachability analysis techniques in rewriting logic for synchronous set relations. More specifically, adapting the rewriting and narrowing based techniques developed in [10], seems promising for the analysis of safety properties of synchronous set relations.

Acknowledgments. The authors would like to thank the anonymous referees for their comments, which helped to improve the paper. This work is supported by NASA's Autonomous Systems and Avionics Project, Software Verification Algorithms. The first author has been partially supported by NSF grant CCF 09-05584 and by the National Aeronautics and Space Administration at Langley Research Center under Research Cooperative Agreement No. NNL09AA00A awarded to the National Institute of Aerospace.

References

1. AlTurki, M., Meseguer, J.: Reduction semantics and formal analysis of Orc programs. Electronic Notes in Theoretical Computer Science 200(3), 25–41 (2008); Proceedings of the 3rd International Workshop on Automated Specification and Verification of Web Systems (WWV 2007)
2. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoretical Computer Science 360(1-3), 386–414 (2006)
3. Chira, C., Serbanuta, T.F., Stefanescu, G.: P systems with control nuclei: The concept. Journal of Logic and Algebraic Programming 79(6), 326–333 (2010); Membrane computing and programming
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350, p. 797. Springer, Heidelberg (2007)
5. Dowek, G., Muñoz, C., Rocha, C.: Rewriting logic semantics of a plan execution language. Electronic Proceedings in Theoretical Computer Science 18, 77–91 (2010)
6. Estlin, T., Jónsson, A., Păsăreanu, C., Simmons, R., Tso, K., Verna, V.: Plan Execution Interchange Language (PLEXIL). Technical Memorandum TM-2006-213483, NASA (2006)

7. Lucanu, D.: Strategy-based rewrite semantics for membrane systems preserves maximal concurrency of evolution rule actions. Electronic Notes in Theoretical Computer Science 237, 107–125 (2009); Proceedings of the 8th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2008)
8. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
9. Meseguer, J., Ölveczky, P.: Formalization and correctness of the pals architectural pattern for distributed real-time systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 303–320. Springer, Heidelberg (2010)
10. Rocha, C., Meseguer, J.: Proving safety properties of rewrite theories. Technical report, University of Illinois at Urbana-Champaign (2010), http://dx.doi.org/10.1007/978-3-642-22944-2_22
11. Rocha, C., Muñoz, C., Dowek, G.: A formal library of set relations and its application to synchronous languages. Theoretical Computer Science 412(37), 4853–4866 (2011)
12. Serbanuta, T.: A Rewriting Approach to Concurrent Programming Language Design and Semantics. PhD thesis, University of Illinois at Urbana-Champaign (December 2010), http://hdl.handle.net/2142/18252
13. Serbanuta, T., Stefanescu, G., Rosu, G.: Defining and executing p systems with structured data in k. In: Corne, D., Frisco, P., Paun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2008. LNCS, vol. 5391, pp. 374–393. Springer, Heidelberg (2009)
14. Viry, P.: Equational rules for rewriting logic. Theoretical Computer Science 285(2), 487–517 (2002)

PiStache: Implementing π-Calculus in Scala

Pedro Matiello and Ana C.V. de Melo[⋆]

Department of Computer Science, University of São Paulo,
São Paulo, Brazil
pmatiello@gmail.com, acvm@ime.usp.br

Abstract. π-calculus is a pioneer theory for concurrent and reconfigurable agent systems. It has been widely used as foundation (semantics) for other theories and languages aiming at representing the computational phenomenon of changing systems' behaviour at runtime. In services-oriented applications for example, reconfiguration is highly required due to the needs of configuring systems accordingly to local contexts. Today, a set of researches are devoted to extending π-calculus features to reconcile concepts behind web-services applications. However, a problem still remains: how to simulate π-agents to have insights on the real behaviour of the specified system? The reconfiguration features embedded in π-calculus enrich its expressiveness but impose a more elaborate semantics, making its implementation a challenging task. The current work presents an implementation of all π-calculus core elements with which one can define agents and simulate them. Such implementation is given as a Domain Specific Language (DSL) in Scala.

Keywords: Pi-calculus, Scala, reconfiguration.

1 Introduction

Dynamic reconfiguration of processes is one of the key-points for many of recent applications based on Web-services, cloud, grid and autonomic computing. Computations in all these scenarios require systems to be reconfigured depending on local contexts. Despite promoting features for global computation, reconfiguration brings challenging issues regarding complexity of systems design: how to predict systems behaviours if they can be reconfigured dynamically.

Formal analysis is employed to predict undesired systems behaviours via formal verification or simulation and can, for instance, be used to analyse behaviours of reconfigurable systems. π-calculus is a pioneer theory for reconfigurable concurrent computation by the means of formal modeling. It provides a simple, yet expressive, foundation for describing process interaction and reconfiguration uniquely through communication, in the form of message passing. Due to its power, it has inspired many other calculi and languages to represent

[⋆] This project has been co-funded by the National Council for Scientific and Technological Development (CNPq - Brazil) and the Ministry of Education Research Agency (CAPES - Brazil).

scenarios on various application domains, including Web-services [2,1] and grid services [15] computing.

Analysing the actual systems behaviours described in π-calculus requires implementation of the calculus. Its reconfiguration features, focused on message passing, improve expressiveness but need an intricate semantics, making its implementation a non-trivial task. For instance, implementation of nondeterministic composition and choice operators under reconfiguration may explode the system behaviour into many threads. Then, a number of thread explosion mechanisms must be implemented together with the possible communication synchronisations, and facilities (or difficulties) to provide these sophisticated mechanisms depends on the chosen programming language. Scala [7] is a general purpose programming language that compile to Java bytecodes. It integrates features from both object-oriented and functional programming. Also, it provides a sophisticated static type system, but still manages to conciliate this with a flexible and concise syntax, and is used in the current project to implement π-calculus core elements.

Being such a simple and expressive language for expressing concurrency and runtime reconfiguration, a set of concrete implementations of π-calculus have already been proposed. They implement different fragments of the calculus and target different uses. PiLib [3] is an implementation of π-calculus in Scala introduced by Vincent Cremet and Martin Odersky. It is part of Scala's standard library and provides a monadic, synchronous and typed version of the calculus with guarded sums. Different from PiLib, Pict [11,12] is a programming language based on the π-calculus computation model. In this implementation, the calculus is polyadic, asynchronous, typed and without summation. It provides a syntax similar to the original calculus and some predefined processes and links for convenience. Kroc (Kent Retargetable occam-pi Compiler) [14] is an occam-pi compiler for Intel 386 and compatible processors. The occam-pi language itself is a variation of the original occam language built on the top of π-calculus, instead of the Communicating Sequential Processes (CSP) [4]. occam-pi implements a typed, synchronous and monadic (including data structures) variant of the calculus. Guarded sums and process mobility (between different machines) are supported.

This paper presents PiStache[1] (Pi-calculus/Scala tache): an implementation of π-calculus as a Domain Specific Language (DSL)[2] hosted in Scala.

2 Background

2.1 π-Calculus

π-calculus, introduced by Milner, Parrow and Walker in [6], is a process calculus for concurrent computation with dynamic reconfiguration. Agents (also called

[1] `http://code.google.com/p/pistache/`

[2] A Domain Specific Language is a programming language designed to a specific purpose. A DSL is said to be internal (or embedded) when it is built within a host language (usually as a library or framework).

processes) communicate by the exchange of names through channels (links). Since channels are treated as names, systems interconnections may change as channel names are passed and shared. This section is a brief introduction to a monadic version of π-calculus. For a more detailed description, one should refer to [6], [9] and [5].

Given a set of *names* $\{x, y, z, ...\}$, a set of *agent identifiers* $\{A, B, C, ...\}$ and a set of *agents* $\{P, Q, R, ...\}$, the syntax of agents is defined as follows:

Definitions: $A(x_1, ..., x_n) \overset{\textbf{def}}{=} P$			**Agents:**	$P ::=$	0	Nil
					$\alpha.P$	Prefixing
					$P + P$	Summation
Prefixes:	$\alpha ::= \bar{y}x$	Output			$P \| P$	Composition
	$y(x)$	Input			$(\nu x)P$	Restriction
	τ	Silent			$[x = y].P$	Match
					$[x \neq y].P$	Mismatch

The operational semantics of process algebras are given by labelled transition systems. Transitions are of kind $P \xrightarrow{\alpha} Q$, for agents ranging over $\{P, Q, ...\}$ and transitions ranging over $\{\alpha, ...\}$. In $P \xrightarrow{\alpha} Q$, P is subject to a labelled transition α leading to Q, and transition rules are:

Prefix	$\dfrac{}{\alpha.P \xrightarrow{\alpha} P}$	Restriction	$\dfrac{P \xrightarrow{\alpha} P', x \notin \alpha}{(\nu x).P \xrightarrow{\alpha} (\nu x).P'}$
Summation	$\dfrac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$	Summation	$\dfrac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$
Match	$\dfrac{\alpha.P \xrightarrow{\alpha} P}{[x=x].P \xrightarrow{\alpha} P}$	Mismatch	$\dfrac{\alpha.P \xrightarrow{\alpha} P, x \neq y}{[x \neq y].P \xrightarrow{\alpha} P}$
Parallel	$\dfrac{P \xrightarrow{\alpha} P', bn(\alpha) \cap fn(Q) = \emptyset}{P \| Q \xrightarrow{\alpha} P' \| Q}$	Parallel	$\dfrac{Q \xrightarrow{\alpha} Q', bn(\alpha) \cap fn(P) = \emptyset}{P \| Q \xrightarrow{\alpha} P \| Q'}$
Communication	$\dfrac{P \xrightarrow{\alpha(x)} P', Q \xrightarrow{\bar{\alpha}u} Q'}{P \| Q \xrightarrow{\tau} P'\{u/x\} \| Q'}$		

Example: The following example, inspired by [9], shall illustrate the calculus. Let C, P and S be agents for a client, a printer and a print server, respectively. The print server and the client share a communication channel a, and the print server and the printer share another communication channel b. The intended interaction is to have S sharing access to P with C, and then have C sending a message to P. Also, after performing their tasks, agents P and S should return to their starting state and the agent C should stop. The agents can be defined as:

$C = a(p).\bar{p}x$
$P = b(y).P$
$S = \bar{a}b.S$

And the entire systems is given by parallel composition of these three agents:

$C|P|S$

2.2 Scala

Scala[3] is a general purpose programming language designed to integrate features of object-oriented and functional programming, and compile to Java bytecodes. Although it is a statically typed language (i.e. values types are known and checked at compile time), it offers some mechanisms to bring as much as possible the conveniences of dynamically typed languages without sacrificing safety features provided by static typing. Here we briefly introduce the main elements of this language used in the present work. Readers are referred to [7] for an extensive guide and those familiar to Java might also find [13] useful.

Variable declaration: Scala supports two types of variables: `val`s, which can be assigned only once, and `var`s, which can be reassigned. For example:

```
val string:String = "Not reassignable"
var string:String = "Reassignable"
```

In most cases, the compiler can infer variables types and the example above can alternatively be written as:

```
val string = "Not reassignable"
var string = "Reassignable"
```

Method definition: Methods are defined with `def` keyword.

```
def max(x:Int, y:Int):Int = {
    if (x > y) x else y
}
```

And, in many cases, the return type can also be inferred by the compiler:

```
def max(x:Int, y:Int) = if (x > y) x else y
```

Class definition: Classes, defined by `class` keyword, can be used to wrap variables and methods.

```
class SpecialInt(int:Int) {
    def isPositive = int >= 0
    def isNegative = int <= 0
}
```

Arguments in the first line belong to the class constructor and are visible to contained methods. Instances of classes are built using the keyword `new`:

```
val number = new SpecialInt(10)
```

Inheritance and Traits: Scala provides two inheritance mechanisms:

1. subclasses:

```
class VerySpecialInt(int:Int) extends SpecialInt(int:Int) {
    def isZero = int == 0
}
```

2. construct `trait`: similar to Java `interface`, supporting method implementations.

[3] http://www.scala-lang.org/

```
trait Person {
    def sleep { Thread sleep 1000 }
    def talk:Unit
}

class NicePerson extends Person {
    def talk { println("Hello") }
}
```

`NicePerson` class is forced by compilers to provide an implementation to method `talk`. Implementation of method `sleep` is, in turn, optional.

Implicit Conversions: If an expression E of type T is expected to be of type S, and T does not extend S, Scala compiler will try to implicitly convert E to type S by using a predefined conversion rule.

The simplest case is converting a value to an expected type on method calls:

```
implicit def Int2String(int:Int) = int.toString

def len(str:String) = str.size
```

With the conversion above in scope, method `len` can be called with an integer argument (and will return the length of its decimal representation).

This mechanism can also be used to `new` methods to existing classes:

```
class SpecialInt(int:Int) {
    def isPositive = int >= 0
    def isNegative = int <= 0
}

implicit def Int2SpecialInt(int:Int) = new SpecialInt(int)
```

When the conversion above is in scope, code like `3.isPositive` will compile as it were in `Int` class.

Pattern Matching: Scala supports *case classes* construct. If a class has a `case` modifier, all its constructor parameters become public class attributes and allow for recursive decomposition combined with *pattern matching*, another feature of the language. Pattern matching affects programs control-flows: it permits multiple type matches and can execute different branches of code, depending on the type of the matched object and of values enclosed by it. The following example illustrates the use of pattern matching on `case` classes:

```
trait Human
case class Man(name:String, age:Int) extends Human
case class Woman(name:String, age:Int) extends Human

def whoIs(human:Human) {
    human match {
        case Man(name, age) => println("His name is " + name)
        case Woman(name, age) => println("Her name is " + name)
    }
}
```

3 PiStache: An Application Programming Interface (API)

PiStache provides an internal domain-specific language for writing π-calculus programs in Scala. The fragment of π-calculus supported is the typed monadic and synchronous version:

Prefixes:	$\bar{y}x \mid y(x) \mid \tau$
Agents:	$0 \mid \alpha.P \mid y(x).P + z(x).Q \mid P \vert Q \mid (\nu x)P \mid [x = y].P \mid [x \neq y].P$
Definitions:	$A(x_1, ..., x_n) \overset{\textbf{def}}{=} P$

All π-calculus operators are implemented but a restriction is imposed to summation: guarded agent summation. First, agents definitions and the core elements are defined (names, actions and channels (links)), followed by the operators.

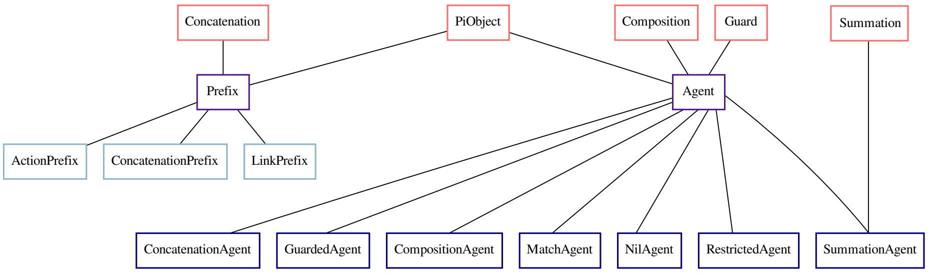


Fig. 1. Pistache architecture

3.1 Core Elements

Agent Definition: The common idiom for defining agents is:

```
val agent = Agent(...)
```

Self-referential agents, used to implement recursive behaviour, require some boilerplate code. In order to satisfy Scala's type checker in compile time, the `Agent` type can not be omitted. Also, to avoid runtime errors due to references to an still non-instantiated object, lazy evaluation[4] must be used.

```
lazy val recursiveAgent:Agent = Agent(...)
```

Agents with arguments can be defined as below:

```
def agentWithArgs(arg1:Type1, ..., argN:TypeN):Agent =
    Agent {
        ...
    }
```

Note that recursive agents substitute for replication operator...
The nil agent is represented by a class named NilAgent.

Names: Names can be used as references to objects. For instance:

```
val name = Name(some_object)
```

[4] Lazy evaluation can be defined by two main traits: "First, the evaluation of a given expression is delayed, or *suspended*, until its result is needed. Second, the first time a suspended expression is evaluated, the result is *memorized* (i.e., cached) so that, if it is ever needed again, it can be looked up rather than recomputed." [8].

The referred object then can be set or changed:

```
name := other_object
```

And retrieved:

```
referred_object = name.value
```

Changing names on referred objects provides reconfiguration of agents when names represent channels: the capability of communicating with other agents changes. This feature, together with the composition operator, makes agents that receive a channel name as input to reconfigure their communication capability from that point on.

Channels (Links): Channels are provided to address communication between agents (processes). Although it is not standard in π-calculus, channels in PiStache are typed (so the values transmitted must be instances of the specified type). `Link` represents channels and can be instantiated:

```
val link = Link[Type]
```

The syntax for sending a name through a channel is:

```
link~name
```

The sent reference can then be received and bound to another name on another agent:

```
link(another_name)
```

Actions: Output and input actions are implemented, respectively, as typed channels sending (`link~name`) and receiving (`link(another_name)`) information. On the other hand, silent actions (τ) do not communicate with the environment, they are performed within the context of a process. These actions are ordinary closures, with no arguments nor returning type, wrapped as an agent by calling `Action`:

```
val silentTransition = Action{ ... }
```

An agent may contain a single silent transition:

```
val silent = Action{ doSomething() }
val agent = Agent(silent)
```

3.2 Operators

Sequence: On the ground level, agents can be made of a sequence the prefixes:

```
val sequentialAgent = Agent(prefix1*...*prefixN)
```

Since silent actions are ordinary prefixes, they can also be concatenated:

```
val silent1 = Action{ doSomething() }
val silent2 = Action{ doSomethingElse() }
val agent = Agent(silent1*silent2)
```

Apart from agents being defined as sequences of prefixes, they can also be defined as prefixed agents. Agents can be guarded by any of the defined prefixes: input, output or silent actions. Although in π-calculus the ordinary concatenation operator is used to guard an agent, PiStache requires the use of a semantically equivalent guard operator.

```
val guardedAgent = Agent { guardPrefix :: Agent }
```

Recursive agents can also be defined along the same lines, enhancing expressiveness to represent infinite behaviours lazyly evaluated:

```
lazy val recursiveAgent:Agent = Agent(prefix1*...*prefixN*recursiveAgent)
```

Guarded Summation: π-calculus summation is a binary operator defined over any two agents. Its semantics is given by a choice of all the first prefixes within agents. Due to efficiency reasons, implementation of summation here is given over guarded instead of ordinary agents. In this particular case, when a summation is executed, only one of its terms will be selected and executed.

```
val summation = Agent {
    val t = Action { ... }
    (y~(x) :: P1) + (y(x) :: P2) + (t :: P3)
}
```

This operation is implemented naively: a list of guarded agents is formed and randomized. Then, the algorithm polls each of the guard actions, attempting to execute it. If it succeeds, the corresponding agent is selected for execution, while the others are discarded. If it fails, the algorithm moves to the next guarded agent until it reaches the end of the list. At this point, it starts over from the beginning. This process is performed until a guard action is executed.

Composition: An agent can also have its behaviour defined as a composition of two other agents running in parallel:

```
val composedAgent = Agent(agent1 | ... | agentN)
```

The late semantics for composition has been implemented here. This means that "handshaking" is provided in communication with the actual names and reconfiguration is realised afterwards.

Having more than one agent running makes communication possible:

```
val square = Link[Link[Int]]

val C = Agent {
    val link = Link[Int]
    val reply = Name[Int]
    val print = Action { println(reply.value) }
    square~link*link~5*link(reply)*print
}

val S = Agent {
    val l1 = Name[Link[Int]]
    val number = Name[Int]
    val calculate = Action {
        number := number.value * number.value
    }
    square(l1)*l1(number)*calculate*l1~number
}
```

When `C | S` is executed, `C` sends the `link` channel name to `S` through channel `square`. Due to reconfiguration, in agent `S`, `link` will be referred to by identifier `l1`. Then, `C` sends through `link` an integer value (5). Finally, `S` uses the same link to send back to `C` the square of the integer sent.

Restriction: To constrain scope of π-calculus agents, the restriction operator must be applied. It is a unary operator whose semantics constrains the use of restricted names to a certain scope. This means that such names cannot be used outside the given scope – they become invalid names. Scope restriction operator is also implemented in PiStache:

```
val agentWithRestrictedName = Agent {
    val restrictedName = Name(...)
    ...
}
```

When restriction is applied, names in `restrictedName` are no longer available outside the scope of agent `agentWithRestrictedName`.

Match: In order to direct the execution flow of programs, `If` match structure is provided. The accepted syntax in PiStache is:

```
val agent = Agent(If (condition) {thenAgent})
```

When executed, if `condition` evaluates to `true`, agent `thenAgent` is to be executed. Otherwise, `agent` halts. Mismatch operator corresponds to match with a negated condition. Then, composition operator can be used together with `If` when both branches of execution are required: either when `condition` evaluates to `true` or `false`:

```
If (condition) {P} | If (!condition) {Q}
```

3.3 Internal Representation

PiStache types are crafted to enforce π-calculus' syntactic rules. Therefore, code written in the syntax presented above is checked at compile time for type errors. It is provided by Scala compiler and refuses meaningless code:

```
val agent = Agent(p1*45.3)
```

For this code, the compiler outputs: `type mismatch; found : Double(45.3) required: pistache.picalculus.Agent`. Avoiding type errors like this is the duty of type systems[5], but this can only be done properly by having types representing domain models as close as possible. Therefore, in order to provide both π-calculus' restrictions and flexibilities, a number of types were implemented:

Type Name	**Description**
PiObject	A trait for tagging all π-calculus' objects. It provides no behaviour.
Concatenation	A trait providing the prefix concatenation operation.
Composition	A trait providing the parallel composition operation.
Summation	A trait providing the summation operation.
Guard	A trait providing the guard operation.
Prefix	A trait for all π-calculus' prefixes.
Agent	A trait for all π-calculus' agents.

[5] "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." [10].

```
trait PiObject

trait Concatenation{ def *(other: => Prefix):ConcatenationPrefix
                     def *(other: => Agent):ConcatenationAgent }

trait Composition  { def |(other: => Agent):CompositionAgent }

trait Summation    { def +(other: => GuardedAgent):SummationAgent }

trait Guard        { def ::(other: => Prefix):GuardedAgent }

trait Prefix extends PiObject with Concatenation

trait Agent  extends PiObject with Composition with Guard
```

The prefixes are provided by the following case classes:

Type Name	Description
ActionPrefix	A case class representing silent transitions.
ConcatenationPrefix	A case class representing the concatenation of two prefixes.
LinkPrefix	A case class representing the an action (input or output) through a link.

```
case class ActionPrefix(val procedure: () => Unit) extends Prefix
case class ConcatenationPrefix(val left: () => Prefix, val right: () =>
    Prefix) extends Prefix
case class LinkPrefix[T](val link:Link[T], val action:ActionType, val name:
    Name[T]) extends Prefix
```

And agents are provided by the following case classes:

Type Name	Description
ConcatenationAgent	A case class representing the concatenation of a prefix and an agent.
GuardedAgent	A case class representing an agent guarded by a prefix.
CompositionAgent	A case class representing the parallel composition of two agents.
SummationAgent	A case class representing the summation of two agents.
MatchAgent	A case class representing an agent conditioned by a match.
NilAgent	A case class representing the null agent.
RestrictedAgent	A case class representing agents with support for restricted names.

```
case class ConcatenationAgent(val left: () => Prefix, val right: () => Agent)
     extends Agent

case class GuardedAgent(val left: () => Prefix, val right: () => Agent)
    extends Agent with Summation

case class CompositionAgent(val left: () => Agent, val right: () => Agent)
    extends Agent

case class SummationAgent(val left: () => Agent, val right: () => Agent)
    extends Agent with Summation

case class MatchAgent(val condition: () => Boolean, val then: () => Agent)
    extends Agent

case class NilAgent() extends Agent

case class RestrictedAgent(val agent: () => Agent) extends Agent
```

Assuming that the programmer made no syntax mistakes, the compiler should be satisfied. Then, at runtime, full π-calculus agents will be built from their atomic parts, as specified by programmers. The following examples illustrate how agents and prefixes are built from concatenation:

```
// suppose  agent Q and prefixes p1, p2, p3 are already defined
val P = Agent(p1*p2*p3*Q)
```

It corresponds to agent $P = p_1.p_2.p_3.Q$ written as PiStache object. Since method * is in an infix form, evaluation of expressions involving it first evaluates the left and then the right-hand sides. It first checks if there is a prefix on its left-hand side and then takes as argument the prefix on its right-hand side and returns a specific type of prefix: `ConcatenationPrefix` if it is a prefix; or `ConcatenationAgent` if it is an agent. So, the first invocation of * has `p1` on the left-hand side, `p2` on the right-hand side, and produces `ConcatenationPrefix(p1, p2)` as a result. The second invocation has `ConcatenationPrefix(p1, p2)` on the left-hand side, `p3` on the right-hand side, and produces `ConcatenationPrefix(ConcatenationPrefix(p1, p2), p3)` as result.

Agents containing composition operator have a similar evaluation. The only difference is that the type wrapping agents on the left and on the right are `CompositionAgent` instead. So, the parallel agent `Agent(P | Q | R)` is built as `Composition-Agent(CompositionAgent(P, Q), R)`.

Now, a last case must be considered:

```
// suppose prefixes p1, p2, p3 and agent Q are already defined
val P = Agent(p1*p2*p3 | Q)
```

As seen before, `p1*p2*p3` is of type `ConcatenationPrefix`, which does not provide a | method for parallel composition. The whole assignment accounts for agent $P = p_1.p_2.p_3|Q$ in pen-and-paper π-calculus. At a first glance, it is a valid expression (as abbreviation of $P = p_1.p_2.p_3.0|Q$) and one would expect that the code above would compile fine. In order to do so, PiStache uses an implicit conversion to produce an agent from concatenated prefixes. More precisely: from `p1*...*pN` it will produce `ConcatenationAgent( p1*...*pN, NilAgent)` which accounts for the explicitly terminated agent $p_1...p_n0$ in π-calculus.

3.4 Execution Model

Since agent objects are simple data structures, devoid of any behaviour, an external mechanism is needed to turn all these objects into an actual program. This task is performed by a specification runner, which takes an agent as argument, interprets its structure and executes the appropriate actions. This approach, implemented in PiStache, provides some flexibility, allowing for implementations of different runners, tailored for different execution environments or π-calculus variants.

The thread-based runner, namely `ThreadedRunner` in PiStache, uses ordinary system threads to execute π-calculus agents concurrently (basically, each agent has its own thread, and agents execution can be carried by as many cores are available in the machine). Communication between agents is synchronous and it is prepared to handle any expression accepted by the API (i.e. any compilable expression). In order to understand this implementation, both the general execution and message passing mechanisms must be observed.

Two operations are possible on channels, output and input, and agents communication is performed through exchanging information between complementary channels in a synchronous manner (message passing). Agents attempting to send a message are blocked until a message is actually received by some other agent, and agents attempting to receive a message are blocked until a message is actually sent.

In a parallel composition, for each channel, both the output and input sides share a lock for mutual exclusion in order to keep at most one thread active at a time. When a thread is set to wait, its execution will pause until it is awaken by a signal. Different channels use different lock objects in order to avoid interference. For agents in a summation, however, at most one term of a summation is to be executed, the execution of guard prefixes must be attempted sequentially (and not concurrently) until one of them succeeds. Then, a variation of mutual exclusion with guard input/output prefixes has been implemented to enforce that the appropriate preconditions are satisfied before calling the basic input or output algorithms. The guarded variants report their success status as return value, as this information is required by the runner in order to determine the execution flow.

Producing threads is an expensive process. Although this cost is negligible in long-lived agents, it takes a quite significant share of the lifetime of processes that exist for only a few hundred milliseconds. If too many of these short-lived agents are spawned in a short amount of time, the runner's performance may be severely harmed. The costs of new agents, still, can be minimized by the reuse of threads. Java platform provides a cached thread pool[6], which can be used to start threads. When a new thread is requested, the pool will reuse one of the cached threads, if one is available; otherwise it will produce a new one. Finished threads are cached and, if not used within 60 seconds, discarded (so memory can be released). This resource has also been implemented in PiStache.

4 API Usage

For the purpose of demonstrating the viability of PiStache as a platform for developing concurrent applications, some small programs were written.

Client-Server example: This is an implementation of client-server [9], in which three agents (C, S and P) communicate in order to print a message. The π-calculus version is:

$$C = (\nu p)(\nu x)a(p).\bar{p}x \qquad S = \bar{a}b.S \qquad P = (\nu y)b(y).P$$

The PiStache version below presents the same semantics. `Server` (agent S) sends a link name to `Client` (agent C) that identifies communication with `Printer` (agent P). `Client` will then send a message to `Printer`, which will display it on the screen.

```
object Printserver {
    def main (args:Array[String]) {

        val sl = Link[Link[String]]
        val pl = Link[String]
        val Client = Agent {
```

[6] http://download.oracle.com/javase/6/docs/api/java/util/concurrent/Executors.html

```
            val l = Name[Link[String]]
            sl(l) * l~"Hello, world!"
        }
        lazy val Server:Agent = Agent {
            sl~pl*Server
        }
        lazy val Printer:Agent = Agent {
            val msg = Name[String]
            val act = Action { println(msg.value) }
            pl(msg) * act * Printer
        }
        new ThreadedRunner(Client | Server | Printer) start
    }
}
```

It is worth to note the explicit declaration and typing of names (including the communication links), which is not required in pure π-calculus.

HTTP Server example: A tiny HTTP Server, capable of serving files in the execution directory, was written and is presented here in a simplified form. First, this application needs to perform communication through TCP sockets. The class below takes, as constructor arguments, a socket and two π-calculus channels. It has a method returning an agent that behaves like an interface between application and sockets.

```
class PiSocket(socket:Socket, send:Link[String], recv:Link[String]) {

    private val wBuf = Name[String]
    private val rBuf = Name[String]
    private val writeToSocket = Action { send value of wBuf through socket }
    private val readFromSocket = Action { receive a line from socket and
         store on rBuf }
    private val closeSocket = Action { close socket }
    def agent() = Agent {
        lazy val writer:Agent = send(wBuf) * (If (wBuf != null) {
             writeToSocket * writer} | If (wBuf == null) {closeSocket})
        lazy val reader:Agent = readFromSocket * recv~rBuf * reader
        writer | reader
    }
}
```

Given a socket and two channels (e.g. `send` and `recv`), the call bellow will return the desired agent:

```
new PiSocket(socket, send, recv).agent
```

This agent can read a message from channel `send`, and send this message through the socket if it is not null (otherwise, the socket will be closed). Similarly, it will receive a message from the socket and send it through `recv`. Therefore, a second agent can send and receive a message through the socket (by sending it through `send` and receiving through `recv`). Now, having this class, the actual server can be presented. It is composed of agents:

`serverAgent`	Accepts incoming connections and spawns other agents to handle them
`handlerAgent`	Spawns an agent for interfacing with the socket and an agent to receive the HTTP request
`loop`	Receives the request and, depending on its validity, continues as one of the agents below
`okAgent`	Sends a header reporting success followed by the requested file
`errAgent`	Sends a header reporting failure

```
def main(args:Array[String]) {

    val serverSocket = new ServerSocket(8080)
    lazy val serverAgent:Agent = Agent {
        var requestSocket:Socket = null
        val accept = Action { requestSocket = serverSocket.accept }
        accept * (serverAgent | handlerAgent(requestSocket))
    }
    def handlerAgent(socket:Socket) = {
        val send = Link[String]
        val recv = Link[String]
        var fileData:String = null
        val requestSocketAgent = new PiSocket(socket, send, recv).agent

        lazy val loop:Agent = Agent {
            val buffer = Name[String]
            val parse = Action {
                parse request and fill the contents of the requested file in
                     fileData
            }
            recv(buffer) * parse * (
                If (buffer.value != null) {loop} |
                If (buffer.value == "" && fileData != null) {okAgent(send,
                     fileData)} |
                If (buffer.value == "" && fileData == null) {errAgent(send)})
        }
        requestSocketAgent | loop
    }
    def okAgent(send:Link[String], extension:String, data:String) = Agent {
        send~HEADER_OK * send~data * send~null
    }
    def errAgent(send:Link[String]) = Agent {
        send~HEADER_ERR * send~null
    }
    new ThreadedRunner(serverAgent).start
}
```

Of course, the code above is lacking many details, including error handling and the relevant imports. The full code is available online[7].

5 Conclusions

Process calculi were developed to express and reason about sets of independent processes and their interactions through mechanisms of message passing. π-calculus is the pioneer member of this family that improve the previous calculi with reconfiguration degree. Although originally developed as a foundation for specification, a number of implementations were created (a few of them being presented earlier in this document). These implementations have demonstrated the feasibility of π-calculus reconfigurable concurrency model, not only as a formal specification tool, but also in actual software programming.

PiStache is a π-calculus implementation similar to PiLib, being written as a domain specific language hosted in the general-purpose language Scala. Still, both implementations play different roles. PiLib offers a more fluid and comfortable interface for Scala programmers, since π-calculus is part of Scala standard implementation. On the other hand, π-calculus elements are not clearly distinguished from Scala itself in PiLib implementation. In PiStache, these interface

[7] http://bitbucket.org/pmatiello/pihttpd

conveniences are sacrificed to keep it syntactically closer to the pen-and-paper calculus, in which every single element of the calculus can be assessed. This compromise is made both to allow for a more direct translation from specifications into actual programs, covering an educational expectation, and for research interest in the implementations of π-calculus verification techniques. Besides that, the execution mechanisms differ in both implementations, with PiStache working atop an explicit abstract syntax tree. This structure can be easily exploited by alternative execution mechanisms implementing different variants of the calculus or even by tools focused on the verification of certain properties in the specifications.

Regarding Kroc and Pict, programming languages developed on the top of π-calculus semantics, there are differences beyond the programming interface. PiStache is hosted in Scala and, in consequence, it exists within the Java ecosystem. This is a very broad and rich environment, and allows for the luxury of integrating with a number of other libraries.

The small HTTP Server presented as proof of concept application upholds the stated perception that π-calculus can contribute to the development of real programs. The simple and expressive mechanics of the calculus, arguably, made the task easier by providing adequate abstractions for reasoning about the problems in question. Also, the treatment of formal expressions as executable code narrows the gap between specifications and implementations, and does so without the use of automatic code generation tools.

The realm of sequential programs has been positively and significantly impacted by the introduction of several abstractions, often packed in paradigms such as structured and object-oriented programming, and in collections of data types. These abstractions facilitated the reasoning on these types of programs by encapsulating low-level details behind more understandable symbols and operations. A similar breakthrough is yet to happen for concurrent programs. Still, this project has been developed under the understanding that the π-calculus might contribute to this goal.

References

1. Bengtson, J., Johansson, M., Parrow, J., Victor, B.: Psi-calculi: a framework for mobile processes with nominal data and logic. Logical Methods in Computer Science 7(1) (2011)
2. Carbone, M., Honda, K., Yoshida, N., Milner, R.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
3. Cremet, V., Odersky, M.: Pilib: A hosted language for pi-calculus style concurrency. In: Lengauer, C., Batory, D.S., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 180–195. Springer, Heidelberg (2004)
4. Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM (1978)
5. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press (1999)

6. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Part I. I and II. Information and Computation 100 (1989)
7. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-step Guide. Artima Incorporation (2008)
8. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
9. Parrow, J.: An Introduction to the pi-Calculus. In: Handbook of Process Algebra, pp. 479–543. Elsevier (2001)
10. Pierce, B.: Types and Programming Languages. MIT Press (2002)
11. Pierce, B.C., Turner, D.N.: Pict: A Programming Language Based on the Pi-Calculus. In: Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press (1997)
12. Pierce, B.C., Turner, D.N.: Pict: a programming language based on the Pi-Calculus, pp. 455–494. MIT Press, Cambridge (2000), http://portal.acm.org/citation.cfm?id=345868.345924
13. Schinz, M., Haller, P.: A Scala Tutorial for Java Programmers (2009), http://www.scala-lang.org/node/198
14. Welch, P., Barnes, F.: Communicating Mobile Processes: introducing occam-pi. In: 25 Years of CSP. Springer, Heidelberg (2005)
15. Zhou, J., Zeng, G.: Describing and reasoning on the composition of grid services using pi-calculus. In: International Conference on Computer and Information Technology, p. 48 (2006)

Sound and Complete Abstract Graph Transformation

Dominik Steenken, Heike Wehrheim, and Daniel Wonisch

Universität Paderborn,
Institut für Informatik,
33098 Paderborn, Germany
{dominik,wehrheim,dwonisch}@mail.upb.de

Abstract. Graph transformation systems (GTS) are a widely used technique for the formal modeling of structures and structure changes of systems. To verify properties of GTS, model checking techniques have been developed, and to cope with the inherent infinity arising in GTS state spaces, abstraction techniques are employed.

In this paper, we introduce a novel representation for abstract graphs (which are shape graphs together with *shape constraints*) and define transformations (execution steps) on abstract graphs. We show that these abstract transformations are sound and complete in the sense that they capture exactly the transformations on the concrete graph level. Furthermore, abstract transformation can be carried out fully automatically. We thus obtain an effectively computable "best transformer" for abstract graphs which can be employed in an abstraction-based verification technique for GTS.

1 Introduction

Today, graphs and graphs transformations are an established formal modeling technique applied in lots of different areas (e.g. business processes [8], refactorings [5] or model transformations [11]). On the one hand, graphs naturally come into play in a model driven software design (since metamodel instances are graphs); on the other hand, graphs are clearly the adequate formalism to describe structural relationships between system components (e.g. the system architecture). In these approaches graph transformation rules (describing modification of graphs) model system steps or changes of the system structure.

As with all formal models, *verification* is also an issue for graph transformation systems (GTSs). We might for instance be interested in proving that certain erroneous structures never arise or that finally specific stable structures are reached. Model checking techniques for GTSs generally follow the idea of generating the whole state space of a GTS, i.e., the set of all graphs reachable from some initial graphs via the application of transformation rules (see e.g. Groove [14]). However, the state space easily gets infinite, for instance, once we have a rule generating new nodes of the graph[1]. Therefore, a number of

[1] If this rule is infinitely often applicable.

approaches for the verification of GTSs try to avoid the construction of the complete set of reachable graphs (e.g., [1,4,20]). Among these are in particular *abstraction* techniques ([3,4,15]); a principle, which our own approach follows as well ([21]). Abstractions for GTSs work on *abstract graphs* instead of concrete representations. In abstract graphs, nodes are not exactly represented but may be *summarized.* A set of nodes can be summarized into one when all nodes are similar; similarity being defined in different ways, e.g. by node types or by a node's neighborhood. Our approach to summarization is inspired by shape analysis techniques for programs ([19]) and thus abstract graphs are also called *shape graphs.* In addition, so called instrumentation predicates on shape graphs help to keep further information about concrete graphs. Instrumentation predicates are an important concept for the abstraction technique as they can tune the verification process towards specific properties.

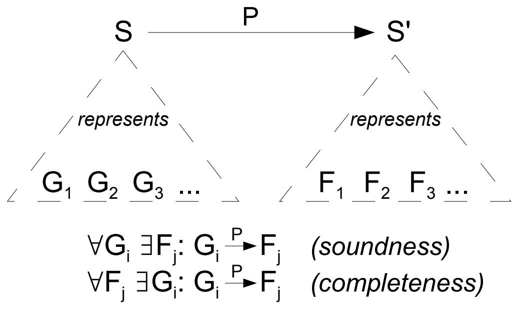

S
P
S'
represents
represents
G1 G2 G3 ...
F1 F2 F3 ...
∀Gi ∃Fj: Gi –P→ Fj (soundness)
∀Fj ∃Gi: Gi –P→ Fj (completeness)

Fig. 1. Soundness and completeness of abstract graph transformation

Every shape graph represents a (potentially infinite) number of concrete graphs (via an embedding). For this technique we thus also need to lift concrete graph transformation rules to abstract graphs, in order to obtain a notion of *abstract* (or shape) *graph transformation.* There are basically two (sometimes contradictory) requirements on this definition: on the one hand abstract transformation should be as precise as possible, on the other hand we have to make an (over)approximation of the concrete transformations as to get only finitely many different abstract graphs. In this paper, we solely look at the first issue, i.e., we develop a notion of abstract graph transformations which precisely reflect concrete transformations. By precise reflection we understand the following two properties (see Figure 1): (1) whenever a concrete graph F_j is reached by applying transformation rule P to a concrete graph G_i, then application of P on an abstraction S of G_i should give us a shape graph S' which represents F_j (*soundness*), and – vice versa – (2) whenever we have a concrete graph F_j represented by S' and S' is reached by applying rule P to S, then we find a graph G_i represented by S such that F_j is reached from G_i by P (*completeness*). When abstract graph transformations are sound, safety properties proven on abstract GTS hold on the concrete GTS as well. Furthermore, completeness guarantees that whenever an error is found in the abstract GTS, the counter example need not be checked for validity on the concretization anymore[2]: every abstract path to an error has a valid concretization.

The search for the "best" abstract transition relation (or abstract transformer [7]) is also an issue for other abstraction techniques [18], like for instance predicate abstraction for software verification. The best abstract transformer often does not immediately give rise to a practical verification method as the set of reachable abstract states might still be infinite in this case. Thus, the best transformer is the ideal which we strive to achieve.

In this paper, we propose a sound and complete definition of abstract graph transformation which defines this ideal, best transformer on shape graphs. In our approach, shape graphs still contain tunable predicates alike instrumentation predicates, here called *shape constraints*. This is mandatory for the verification process as it enables us to keep specific information about concrete graphs in the abstraction. In contrast to instrumentation predicates, shape constraints can however be fully automatically updated during abstract transformations. The abstract transformer thus can (and has been) implemented, and there is no further manual definition from users necessary. We have integrated this new approach into our GTS verification tool (which has been using instrumentation predicates with manually specified update formulas before). An evaluation has shown that the new approach can significantly improve the precision of the analysis. However, when using this ideal abstract transformer the verification procedure does not always terminate. Further work has to show how to find a good compromise between precision and the need for a terminating verification technique. In this paper, we concentrate on the theoretical concept of a sound and complete, and thus best abstract transformation.

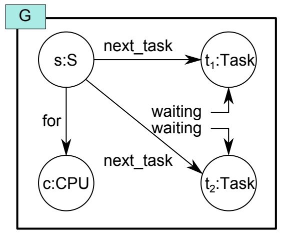
2 Background

We start with a brief introduction to graphs, graph transformation rules and shape graphs. In general, a graph consists of a set of nodes and edges between nodes, edges potentially being labeled. Nodes represent entities of the modeled system and edges some relationships between these entities. The labeling serves as a way of naming relationships. In Figure 2a we see a graph consisting of a scheduler, a CPU and two tasks. In our framework, we represent graphs as logical structures.

Definition 1. *A* graph *G is a logical structure over the logic $\mathcal{I} = \{0, 1\}$, i.e. $G = (U, \mathcal{P}, \iota)$ consisting of a set of nodes (or* individuals*) U, a set of labels (or* predicates*) $\mathcal{P}$ with different arity, and an interpretation function (defining edges) $\iota : \mathcal{P}_k \rightarrow (U^k \rightarrow \mathcal{I})$. Here, $\mathcal{P}_k$ is the set of k-ary predicates.*

As we see here, the definition of edges via interpretations for predicates (label names) gives us the possibility of also having unary (or even n-ary) edges besides

[2] Unless one is interested in reachability from a particular initial state, not a set of initial states.



(a)

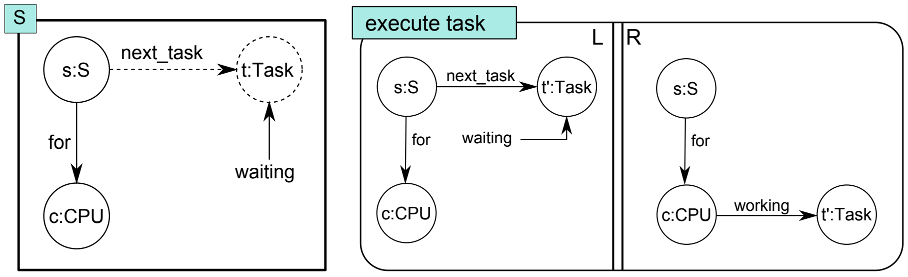
$U = \{s, c, t_1, t_2\}$

$\mathcal{P} = \{for, next\_task, waiting, S, CPU, Task\}$

ι	$waiting$	S	CPU	$Task$	for	s	c	t_1	t_2	nt	s	c	t_1	t_2
s	0	1	0	0		0	1	0	0		0	0	1	1
c	0	0	1	0		0	0	0	0		0	0	0	0
t_1	1	0	0	1		0	0	0	0		0	0	0	0
t_2	1	0	0	1		0	0	0	0		0	0	0	0

(b)

Fig. 2. Graph describing a situation in a scheduling system and its formal representation as logical structure



(a) Shape graph (b) Graph transformation rule

Fig. 3. Start shape graph representing the graph of Figure 2a and a graph transformation rule *Execute Task* that assigns a tasks marked as next task to the CPU

normal ones. Figure 2b gives the logical structure for the graph G from Figure 2a. Here, we see that for instance the types of nodes (task, scheduler etc.) appear as unary predicates.

Graphs are logical structures over boolean logic; an edge is either existing or not existing. In the abstraction we collapse sets of nodes into so-called *summary* nodes and thus lose information about the precise location of edges. This uncertainty about edges is represented by a third truth value, denoted $\frac{1}{2}$. It denotes a kind of "don't know" or "maybe" value. Formally, we use the (strong) three-valued Kleene logic $\mathcal{K}$ (see [9]) for this purpose. A *shape graph* is thus simply a logical structure over $\mathcal{K}$. One particular predicate called sm is in the set of predicates $\mathcal{P}_1$ of all shape graphs. It is used for marking summary nodes. Summary nodes are drawn as dashed circles; maybe edges as dashed lines. The shape graph in Figure 3a thus represents an abstraction of the concrete graph in Figure 2a. Here, all task nodes are summarized by node t. This representation relation is formally defined by embedding relations between logical structures.

Definition 2. *A graph* $G = (U^G, \mathcal{P}, \iota^G)$ *is* embedded *in a shape graph* $S = (U^S, \mathcal{P}, \iota^S)$ *wrt. to the* embedding function $f : U^G \to U^S$*, denoted by* $G \sqsubseteq_f S$*, if*

Table 1. Truth tables for $\wedge$, $\vee$, and $\neg$ in Kleene logic

$\wedge$	0	$\frac{1}{2}$	1
0	0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$
1	0	$\frac{1}{2}$	1

$\vee$	0	$\frac{1}{2}$	1
0	0	$\frac{1}{2}$	1
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1
1	1	1	1

	$\neg$
0	1
$\frac{1}{2}$	$\frac{1}{2}$
1	0

– *for each predicate* $p \in \mathcal{P}$ *and individuals* $u_1, u_2 \in U^G$ *we have* $\iota^G(p)(u_1, u_2) \sqsubseteq \iota^S(p)(f(u_1), f(u_2))$[3]*, and*
– *for each* $v \in U^S$ *such that there are two individuals* $u_1, u_2 \in U^G$ *with* $f(u_1) = f(u_2) = v$ *we have* $\iota^S(sm)(v) = \frac{1}{2}$,

where $l_1 \sqsubseteq l_2$ *holds for logical literals* l_1, l_2 *if* l_2 *is more abstract than* l_1[4].

Intuitively, the embedding function tells us which concrete node is represented by which abstract node.

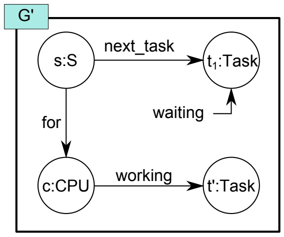
Next, we are interested in modifications of graphs, i.e. graph transformation rules. These rules express the possible steps of our system and tell us how the structure evolves. A graph transformation rule $P = \langle L, R \rangle$ consists of a *left hand side* L and a *right hand side* R. Both are ordinary graphs, i.e. two-valued structures. Figure 3b shows a rule which assigns one of the tasks marked as next task to the CPU. A graph transformation rule P can be *applied* to a graph G if the left hand side can be found in G. Its application replaces the left hand side L by the right hand side R. Formally, we have to find an injective function $m : U^L \to U^G$, called *matching*, such that for every binary predicate $p \in \mathcal{P}$ and every $u_1, u_2 \in U^L$ we have $\iota^G(p)(m(u_1), m(u_2)) = 1$ if $\iota^L(p)(u_1, u_2) = 1$, where $L = (U^L, \mathcal{P}, \iota^L)$. This application condition for rules can also be expressed as a first-order predicate logic formula:

$$\varphi_P = \underbrace{\bigwedge_{\substack{u_1, u_2 \in U^L \\ \iota^L(p)(u_1, u_2) = 1}} p(u_1, u_2)}_{\text{edges}} \wedge \underbrace{\bigwedge_{\substack{u_1, u_2 \in U^L \\ u_1 \neq u_2}} \neg(u_1 = u_2)}_{\text{injectivity}}$$

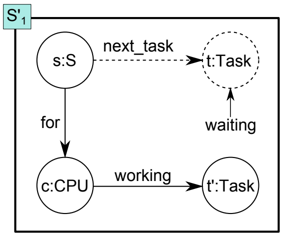
The matching m gives us an assignment of the U^L nodes to nodes of the graph G. From the point of view of the graph G, the elements from U^L play the role of free variables in the formula φ_P. Thus a matching m determines an assignment to the free variables such that evaluation of the formula in the logical structure of G yields true, which is denoted as $[\![\varphi_P]\!]^G_m = 1$.

For a graph transformation rule $P = \langle L, R \rangle$ and a graph G (to which P can be applied), we define the result of applying P to G as the graph G' we get by replacing the left hand side L in G with the right hand side R. Formally,

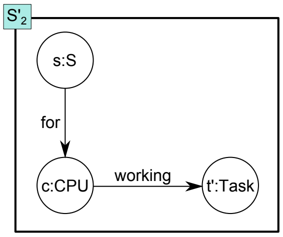
[3] For simplicity we restrict ourselves to binary predicates in our definitions.
[4] That is, $0 \sqsubseteq 0$, $1 \sqsubseteq 1$, $0 \sqsubseteq \frac{1}{2}$, $1 \sqsubseteq \frac{1}{2}$, and $\frac{1}{2} \sqsubseteq \frac{1}{2}$.



(a) Resulting graph



(b) Resulting shape graph (t kept)



(c) Resulting shape graph (t removed)

Fig. 4. Results of applying *Execute Task* to the graph of Figure 2a and the shape graph of Figure 3a, respectively

let $E^- = \{(u_1, p, u_2) \mid u_1, u_2 \in U^L \wedge \iota^L(p)(u_1, u_2) = 1 \wedge ((u_1, u_2 \in U^R \wedge \iota^R(p)(u_1, u_2) = 0) \vee u_1 \notin U^R \vee u_2 \notin U^R)\}$ denote the set of edges that are removed by P and $E^+ = \{(u_1, p, u_2) \mid u_1, u_2 \in U^R \wedge \iota^R(p)(u_1, u_2) = 1 \wedge ((u_1, u_2 \in U^L \wedge \iota^L(p)(u_1, u_2) = 0) \vee u_1 \notin U^L \vee u_2 \notin U^L)\}$ the set of edges that are added by P. Furthermore, we extend matchings $m : U^L \to U^G$ to $\widehat{m} : U^L \cup U^R \to U^G$ be letting $\widehat{m}(u) = u$ for $u \in U^R$. Then the result of applying $P = \langle L, R\rangle$ to $G = (U^G, \mathcal{P}, \iota^G)$ wrt. some matching m is a graph $G' = (U^{G'}, \mathcal{P}, \iota^{G'})$, where $U^{G'} = (U^G \setminus m(U^L)) \cup \widehat{m}(U^R)$ and for each (binary) predicate p and each pair of individuals $u_1, u_2 \in U^{G'}$:[5]

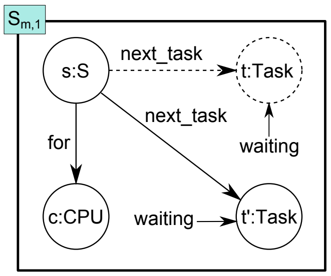
$$\iota^{G'}(p)(u_1, u_2) = \begin{cases} 0 & \text{if } (u_1, p, u_2) \in m(E^-) \\ 1 & \text{if } (u_1, p, u_2) \in \widehat{m}(E^+) \\ \iota^G(p)(u_1, u_2) & \text{otherwise.} \end{cases}$$

We write $G \xrightarrow{P,m} G'$ if G' is the result of applying P to G wrt. the matching m, and $G \xrightarrow{P} G'$ if there is some matching m such that $G \xrightarrow{P,m} G'$. Figure 4a shows the result of applying rule *Execute Task* to the graph of Figure 2a using matching $\{s \mapsto s, c \mapsto c, t' \mapsto t_2\}$.

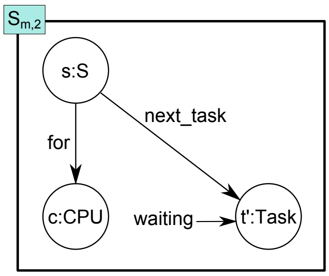
For our abstraction technique, we need to lift transformation rules to the level of shape graphs. To this end, we first of all lift the application formula φ_P to shape graphs. For this, we simply add one new conjunction which formalizes the fact that the matching should not map left hand side nodes to summarized nodes: $\bigwedge_{u \in U^L} \neg sm(u)$. We let φ_P denote this formula as well[6] (see Table 1 for the truth tables of $\wedge$, $\vee$, and $\neg$ in Kleene logic). If $[\![\varphi_P]\!]^S_m = 0$, we cannot apply P to S, if it is 1 we can. However, with shape graph we now also have the case that $[\![\varphi_P]\!]^S_m = \frac{1}{2}$. Intuitively, in this case, P can be applied to some (but possibly not all) of the graphs represented by S. Obviously, we only want to apply P to those graphs on which P can be actually applied. We thus *materialize* shape

[5] Here, m is extended to edge sets via $m(u_1, p, u_1) = (m(u_1), p, m(u_2))$.

[6] This conjunct could have already been added to φ_P assuming that in an ordinary graph G all nodes are not summarized.



(a) Materialized shape graph (t kept)



(b) Materialized shape graph (t removed)

Fig. 5. Materializations of the shape graph of Figure 3a with respect to *Execute Task*

graphs $S_m \sqsubseteq S$ out of S such that these shape graphs S_m represent exactly those graphs that are represented by S *and* allow the application of P wrt. m.

We construct the set of materializations $mat_m^P(S)$ for a shape graph $S = (U^S, \mathcal{P}, \iota^S)$, a transformation rule $P = \langle L, R \rangle$, and a matching $m : U^L \to U^S$ such that $[\![\varphi_P]\!]_m^S = \frac{1}{2}$ as follows. First, we have to consider those summary nodes in S that occur on the left hand side of P, i.e., we consider $\varGamma(m) := m(U^L) \cap U_{sum}^S$, where $U_{sum}^S \subseteq U^S$ denotes the summary nodes in S. Out of every such node u we have to materialize as many concrete nodes as get mapped to u, such that P can be concretely applied on them. The relationship of these materialized node to other nodes of the shape graph are inherited from the original shape graph. In addition, we have to decide for each summary node in $\varGamma(m)$ whether to keep the node or to remove it. This represents the idea that summary nodes can stand for any number of concrete nodes. Thus we get several materializations of one shape graph, one for every set $I \subseteq \varGamma(m)$, I being those summary nodes which we keep in the materialization.

Definition 3. *Let $S = (U^S, \mathcal{P}, \iota^S)$ be some shape graph, $P = \langle L, R \rangle$ be a transformation rule, and $m : U^L \to U^S$ some matching such that $[\![\varphi_P]\!]_m^S = \frac{1}{2}$. Let $\varGamma(m) := m(U^L) \cap U_{sum}^S$. Then, for each $I \subseteq \varGamma(m)$ the* materialization of S wrt. rule P, matching m, and according to I *is defined as $mat_m^P(S, I) = (U', \mathcal{P}, \iota')$, with*[7]

$$U' = U^S \setminus (m(U^L) \setminus I) \cup U^L$$

and for each (binary) predicate p, letting $\widehat{m} = m \cup id_{U^S}$:

$$\iota'(p)(u_1, u_2) = \begin{cases} 1 & \textit{if } u_1, u_2 \in U^L \land \iota^L(p)(u_1, u_2) = 1 \\ \iota(p)(\widehat{m}(u_1), \widehat{m}(u_2)) & \textit{otherwise.} \end{cases}$$

$$\iota'(sm)(u) = \begin{cases} 0 & \textit{if } u \in U^L \\ \iota(sm)(\widehat{m}(u)) & \textit{otherwise.} \end{cases}$$

[7] W.l.o.g. we assume $U^S \cap U^L = \varnothing$.

The collection of all such shape graphs is then defined as the materialization of S wrt. rule P and matching m*:*

$$mat_m^P(S) = \{mat_m^P(S, I) \mid I \subseteq \Gamma(m)\}$$

Note that the size of mat_m^P can be exponential in the number of nodes in the left hand side of the rule, but is finite. For more details, please refer to [21]. After materialization, we apply the rule like for ordinary graphs. Again, we write $S \xrightarrow{P,m} S'$ if S' is the result of applying P to S wrt. the matching m, and $S \xrightarrow{P} S'$ if there is some matching m such that $S \xrightarrow{P,m} S'$.

In Figure 5 we see the materializations of the shape graph S shown in Figure 3a wrt. *Execute Task* and $m := \{s \mapsto s, c \mapsto c, t' \mapsto t\}$. In this case we have $\Gamma(m) = \{s, c, t\} \cap \{t\} = \{t\}$. Thus, we get two materializations, one for $I = \{t\}$ (Figure 5a) and one for $I = \{\}$ (Figure 5b). The former materialization $S_{m,1}$ reflects the case that the summary node t in S represented more or equal to two tasks, where the latter materialization $S_{m,2}$ reflects the case in which t represented exactly one task. On both materializations we can now apply *Execute Task*, leading to the shape graphs shown in Figure 4b and Figure 4c, respectively.

3 Shape Constraints

As we see in Figure 3a the abstraction loses information about the precise number of nodes and edges in the concrete graph. Our verification might however rely on *some* (but not all) of this information. To this end, we attach further information about the represented concrete graphs to the shape graph. This information comes in the form of logical constraints on the represented graphs and in its basic idea (but not its actual usage) follows the idea of instrumentation predicates in shape analysis ([19]). For our example, we might for instance be interested in showing that the CPU never serves more than two tasks at a time. Hence the number of tasks present in concrete graphs is - to a certain extent - of interest. We thus augment the shape graph in Figure 3a with a shape constraint called *task_leq_2* and attach this constraint to node s. Formally, constraint *task_leq_2* is defined by the following *meaning formula*

$$\alpha(v) := \neg \exists\, t_1, t_2, t_3 : (t_1 \neq t_2) \wedge (t_1 \neq t_3) \wedge (t_2 \neq t_3) \wedge \\ \mathit{next\_task}(v, t_1) \wedge \mathit{next\_task}(v, t_2) \wedge \mathit{next\_task}(v, t_3).$$

We attach this shape constraint to a particular node by means of a *binding* of the free variables of the constraint to nodes in the shape graph, here v to s. In general, we thus have the following definition of shape constraints.

Definition 4. *Let $S = (U, \mathcal{P}, \iota)$ be a shape graph. A tuple (c, a) is called a* shape constraint *for S if*

- *c is a first order predicate logic formula over S and*
- *a is an assignment that maps the free variables of c, $free(c)$, to individuals $u \in U$.*

S fulfills a set of shape constraints Λ_S, denoted by $S \models \Lambda_S$, iff $[\![c]\!]^S_a \neq 0$ for every $(c, a) \in \Lambda_S$.

A shape constraint can thus be fulfilled in a shape graph or not fulfilled. Furthermore, a set of shape constraints attached to a shape graph restricts the set of represented concrete graphs.

Definition 5. *Let S be a shape graph, G an ordinary graph and Λ_S a set of shape constraints for S such that $S \models \Lambda_S$.*

The graph G is embedded *in (S, Λ_S), denoted by $G \sqsubseteq (S, \Lambda_S)$, iff $G \sqsubseteq_f S$ for some embedding function f and $G \models concr(\Lambda_S, f)$ for $concr(\Lambda_S, f) := \{(c, a') \mid (c, a) \in \Lambda_S, a' : free(c) \to U^G, f \circ a' = a\}$. We let $concr(S, \Lambda_S)$ denote the set of concrete graphs embedded in (S, Λ_S).*

Intuitively, whenever a shape constraint is true for a particular node in the abstraction it should also hold for all represented nodes in the concretization.

In the following, abstract states are always given as pairs of shape graph together with a set of shape constraints, (S, Λ_S). This gives us additional information in the abstraction. However, now the question arises how this additional information is updated when rules are applied. Our intention is to achieve a rule update for shape constraints which on the one hand is fully automatic (this is in contrast to instrumentation predicates for which manual update formula need to be given), and on the other hand is sound and complete. The latter property means that – given a shape graph with constraints (S, Λ_S) and a rule P – we want to construct some (S', Λ'_S) such that

$$\forall\, G \in concr(S, \Lambda_S)\, \exists\, F \in concr(S', \Lambda_{S'}) \text{ s.t. } G \xrightarrow{P} F \text{ (Soundness)}$$
$$\forall\, F \in concr(S', \Lambda_{S'})\, \exists\, G \in concr(S, \Lambda_S) \text{ s.t. } G \xrightarrow{P} F \text{ (Completeness)}$$

The basic idea how to guarantee soundness and completeness when updating shape constraints is to alter the set of shape constraints such that no information is lost by the application of a given rule. To get some feeling about how the information contained in the shape constraints can be maintained, consider an update mechanism that just leaves the shape constraints unchanged when applying graph transformation rules. That is, we say $(S, \Lambda_S) \xrightarrow{P} (S', \Lambda_{S'})$ if $S \xrightarrow{P} S'$ and $\Lambda_{S'} = \Lambda_S$. This update mechanism is *unsound*. For example, if S is the shape graph of Figure 3a we could safely (without changing the described set of graphs) add a shape constraint (c, a) to S expressing that the CPU does not execute any task. However, if we then apply *Execute Task* to S and leave the shape constraints unchanged, leading to $(S', \Lambda_{S'})$, we effectively describe an empty set of graphs with $(S', \Lambda_{S'})$ (S' enforces an edge from c to t', while the shape constraints forbid any outgoing edges for c). The problem here is that the shape constraint (c, a) describes a fact that is true in S but not in S'.

On the other extreme, we could construct an update mechanism which drops all shape constraints with each rule application. Such an update mechanism is sound, but obviously *incomplete*, as we lose all information contained in the shape constraints with each rule application.

In conclusion, a sound and complete update mechanism should be somewhere in between "keep all constraints unchanged" and "completely remove all constraints". In fact, our solution works as follows: we iterate over each shape constraint $(c, a) \in \Lambda_S$ and check for each atomic subformula φ in c whether φ refers to an outdated information (outdated because the shape graph changed with the rule application). If the information is not outdated, i.e. still valid, we leave the subformula unchanged. Otherwise, we replace it by a literal which describes the former value of φ in S (i.e., we effectively remove the outdated part from the shape constraint).

As an example, reconsider the shape graph $S_{m,1}$ shown in Figure 5a together the shape constraint $(\alpha_{task\_leq\_2}(v), [v \mapsto s])$, where $\alpha_{task\_leq\_2}$ denotes the meaning formula of *task_leq_2*. Moreover, for simplicity, assume for now that we remove the quantifiers in the shape constraint by explicitly naming candidates for t_1, t_2, t_3, namely, we get the formula

$$\neg(\mathit{next\_task}(v, t_1) \wedge \mathit{next\_task}(v, t_2) \wedge \mathit{next\_task}(v, t_3))$$

together with the assignment

$$[v \mapsto s, t_1 \mapsto t, t_2 \mapsto t, t_3 \mapsto t']$$

as simplified shape constraint. Applying *Execute Task* to S_m we see that the rule changes the interpretation of *next_task* for the individuals s and t' (from 1 to 0). Taking our assignment into consideration, we thus have to replace the subformula $\mathit{next\_task}(v, t_3)$ by the literal 1, thereby at least maintaining the information that t represents at most one task that is marked as next task.

Things get a little more complicated if we take quantifiers into account. In that case variables occurring in predicate formulae like $p(u, v)$ may be *bound*. This makes it impossible to statically deduce whether $p(u, v)$ changes (i.e., refers to an outdated information) or not. Nevertheless, our solution for this is to simply encode the decision whether to replace the predicate formula by a literal or not directly inside the formulae, leading to e.g.

$$(((u = w_1) \wedge (v = w_2)) \rightarrow \iota^S(p)(w_1, w_2)) \wedge (((u \neq w_1) \vee (v \neq w_2)) \rightarrow p(u, v)),$$

for some predicate formula $p(u, v)$ whose interpretation changes for individuals w_1, w_2 when some rule P is applied.

In our example, we transform the shape constraints $(\alpha_{task\_leq\_2}(v), [v \mapsto s]) \in \Lambda_{S_m}$ into the shape constraint (c', a') when applying *Execute Task*, where

$$\begin{aligned} c' := \neg \exists\, t_1, t_2, t_3 : (t_1 \neq t_2) \wedge (t_1 \neq t_3) \wedge (t_2 \neq t_3) \wedge \\ ((((v = s) \wedge (t_1 = t')) \rightarrow 1) \wedge (((v \neq s) \vee (t_1 \neq t')) \rightarrow \mathit{next\_task}(v, t_1))) \wedge \\ ((((v = s) \wedge (t_2 = t')) \rightarrow 1) \wedge (((v \neq s) \vee (t_2 \neq t')) \rightarrow \mathit{next\_task}(v, t_2))) \wedge \\ ((((v = s) \wedge (t_3 = t')) \rightarrow 1) \wedge (((v \neq s) \vee (t_3 \neq t')) \rightarrow \mathit{next\_task}(v, t_3))) \end{aligned}$$

and $a' := [v \mapsto s, s \mapsto s, t' \mapsto t']$. The untransformed shape constraint enforces a maximum number of two tasks that are marked as next tasks by the scheduler. The transformed shape constraint, on the other hand, effectively enforces a maximum number of one task that is marked as next task, which meets the intuition, as the rule removed exactly one marked task.

Summing up our idea, we provide the following formal definition for the automatic transformation of shape constraints.

Definition 6. *Let S and S' be shape graphs and $P = \langle L, R\rangle$ a graph transformation rule that does not add or remove any individuals. Let Λ_S be a set of shape constraints for S and $\Lambda_{S'}$ a set of shape constraints for S'. For some matching $m : U^L \to U^S$, we denote $(S, \Lambda_S) \xrightarrow{P,m} (S', \Lambda_{S'})$ iff $S \xrightarrow{P,m} S'$ and $\Lambda_{S'} = \{(transform(c), a \cup id_{U^{S'}}) \mid (c, a) \in \Lambda_S\}$, where $transform(\cdot)$ is inductively defined as follows:*

Atomic. *For $l \in \{0, \frac{1}{2}, 1\}$: $transform(l) = l$.*
For some binary predicate $p \in \mathcal{P}$ and variables v_1, v_2: Let

$$H := \{(w_1, w_2) \mid w_i \in (U^S \cap U^{S'}) \wedge \iota^S(p)(w_1, w_2) \neq \iota^{S'}(p)(w_1, w_2)\}$$

denote the set of combinations of individuals for which the interpretation of p changes when the rule is applied. We define:

$$\begin{aligned} transform(p(v_1, v_2)) = &\left(\bigwedge_{(w_1,w_2)\in H} \Big(((v_1 = w_1) \wedge (v_2 = w_2)) \to \iota^S(p)(w_1, w_2)\Big)\right) \wedge \\ &\left(\left(\bigwedge_{(w_1,w_2)\in H} \neg ((v_1 = w_1) \wedge (v_2 = w_2))\right) \to p(v_1, v_2)\right) \end{aligned}$$

For an atomic formula $(v_1 = v_2)$ (and variables v_1, v_2):

$$transform((v_1 = v_2)) = (v_1 = v_2)$$

Logical Connectives. *For some formulae φ_1 and φ_2:*

$$\begin{aligned} transform(\varphi_1 \wedge \varphi_2) &= transform(\varphi_1) \wedge transform(\varphi_2) \\ transform(\varphi_1 \vee \varphi_2) &= transform(\varphi_1) \vee transform(\varphi_2) \\ transform(\neg\varphi_1) &= \neg transform(\varphi_1) \end{aligned}$$

Quantifiers. *For some formula φ:*

$$\begin{aligned} transform(\exists\, v : \varphi) &= \exists\, v : transform(\varphi) \\ transform(\forall\, v : \varphi) &= \forall\, v : transform(\varphi) \end{aligned}$$

To simplify the definition we assumed that the given transformation rule does not add and remove any individuals. This is – however – not mandatory. If we allow the graph transformation rule to add and remove individuals, we have to alter e.g. the transformation for existential quantifiers such that we exclude newly added individuals and include removed individuals in the quantification. That is, we get:

$$transform(\exists\, v : \varphi(v)) = \left(\exists\, v : \left(\bigwedge_{w \in U^{S'} \setminus U^S} \neg(v = w)\right) \wedge transform(\varphi(v))\right) \vee \left(\bigvee_{w \in U^S \setminus U^{S'}} transform(\varphi(w))\right)$$

A full definition that takes all sort of rules into account can be found in [22].

4 Soundness and Completeness

Now that we have seen how abstract transformation rules can be defined for shape constraints, we can state our main result about soundness and completeness[8].

Theorem 1. *Let (S, Λ_S), $(S', \Lambda_{S'})$ be shape graphs with shape constraints and let $P = \langle L, R \rangle$ be a graph transformation rule such that $(S, \Lambda_S) \xrightarrow{P} (S', \Lambda_{S'})$. Then the following holds true.*

- *Soundness: $\forall\, G \in concr(S, \Lambda_S)\, \exists\, G' \in concr(S', \Lambda_{S'})$ s.t. $G \xrightarrow{P} G'$, and*
- *Completeness: $\forall\, G' \in concr(S', \Lambda_{S'})\, \exists\, G \in concr(S, \Lambda_S)$ s.t. $G \xrightarrow{P} G'$.*

A central proposition needed for the proof is that the shape constraints Λ_S of a given shape graph S get transformed by a rule application such that the shape constraints Λ_S are fulfilled in S if and only if the transformed shape constraints are fulfilled in the transformed shape graph.

Lemma 1. *Let S be a shape graph. Let Λ_S be a set of shape constraints for S. Let P be a graph transformation rule. Let $(S, \Lambda_S) \xrightarrow{P} (S', \Lambda_{S'})$. We have $S \models \Lambda_S$ if and only if $S' \models \Lambda_{S'}$.*

Proof. By induction over the structure of shape constraints in Λ_S. □

We can now sketch the proof of Theorem 1.

Proof of Theorem 1. We start by sketching the proof of completeness. To this end, we pick some $G' \in concr(S', \Lambda_{S'})$. Let $G' \sqsubseteq_{f'} S'$ and $\Lambda_{G'} := concr(\Lambda_{S'}, f')$ ($G' \models \Lambda_{G'}$). We have to show that there is some $G \in concr(S, \Lambda_S)$ such that $G \xrightarrow{P} G'$. To find such a graph G, we simply backwards apply P to G' to get an unique graph $G \sqsubseteq_f S$ in our DPO approach. Afterwards, we are left to show that $G \models \Lambda_G$ holds for $\Lambda_G := concr(\Lambda_S, f)$. To show this, we apply P onto (G, Λ_G) to

[8] For the theorem we actually assume a DPO-like (double-pushout, [13]) approach for the application of transformation rules rather than the SPO-like (single-pushout) approach as suggested in Section 2.

get $(G', \Lambda_{G'_{aux}})$. While in general $\Lambda_{G'} \neq \Lambda_{G'_{aux}}$, we can find for each $(c, m) \in \Lambda_{G'_{aux}}$ a $(c', m') \in \Lambda_{G'}$ (and vice versa) such that c only syntacticly differs from c' in the free variables introduced with the rule application. Furthermore, each such free variable u in c gets assigned to same individual $u \in U^{G'}$ by m $(m(u) = \mathrm{id}(u) = u)$ as the corresponding free variable u' in c' by m' $(m'(u') = f'^{-1}(u') = u$ [9]). We thus can conclude from $G' \models \Lambda_{G'}$ that $G' \models \Lambda_{G'_{aux}}$ also holds. By Lemma 1 it follows that $G \models \Lambda_G$.

For the soundness we start with a graph $G \in concr(S, \Lambda_S)$ and show that for G' with $G \xrightarrow{P} G'$ we have $G' \in concr(S', \Lambda_{S'})$. Let $G \sqsubseteq_f S$ and $\Lambda_G := concr(\Lambda_S, f)$. We apply P onto G and get $(G, \Lambda_G) \xrightarrow{P} (G', \Lambda_{G'_{aux}})$ with $G' \sqsubseteq_{f'} S'$. By Lemma 1 we can conclude with $G \models \Lambda_G$ that $G' \models \Lambda_{G'_{aux}}$. Again, in general, $\Lambda_{G'_{aux}} \neq \Lambda_{G'}$ for $\Lambda_{G'} := concr(\Lambda_{S'}, f')$. But similar as above, one can show that $G \models \Lambda_{G'}$ follows from $G' \models \Lambda_{G'_{aux}}$. □

For our verification technique, we now get the following property from Theorem 1. We start the construction of the abstract state space with a shape graph together with shape constraints which represent all possible initial graphs, i.e. graphs representing initial states of our system. The state space is then constructed using the abstract transformations. Whenever we reach an abstract state invalidating our (safety) property, we directly get a valid counter example, i.e. by Theorem 1 we know that there is a sequence of concrete steps starting from one of the initial graphs reaching an erroneous concrete graph.

We have integrated shape constraints and their updates into our graph transformation tool which is built on-top of the 3-valued logic engine TVLA [6]. An evaluation shows that the overhead of using shape constraints is almost negligible (for terminating analyses) and that the length and amount of shape constraints usually stay at a manageable size.

5 Conclusion

In this paper we introduced a novel approach for the representation of abstract graphs that relies on shape graphs and shape constraints. We furthermore showed how to fully-automatically apply graph transformation rules to our abstract graphs such that no additional information is lost. That is, we can efficiently compute the best abstract transformer in our abstract domain and the abstract domain is expressive enough such that an exploration of the abstract state space based on the best abstract transformer is both sound and complete. In consequence every counterexample in the abstract state space we find is guaranteed to have a concrete counterpart.

Related Work. Several approaches for the construction of abstract transformers can be found in literature. Graf and Saïdi showed in [10] how to construct best abstract transforms for Cartesian predicate abstractions using a theorem

[9] The inverse of the embedding function f'^{-1} is well-defined for these individuals as they are non-summarizing (which is guaranteed by the materialization).

prover. In [2] the abstract domain of [10] is generalized to boolean predicate abstraction and several abstract transformers differing in its precision and efficiency are suggested. [17] further generalizes these results in that an approach for the computation of best abstract transforms for arbitrary finite-height abstract domains (lattices) is introduced.

More closely to graph transformation systems, in [4], a method for automatic abstraction of graphs is introduced. Intuitively speaking, nodes are identified if their neighborhood of radius $k \in \mathbb{N}$ is the same. Following this approach, the authors describe a fully automatic method for the verification of GTSs. In the approach, the transformation of abstract graphs is performed in three stages. First, the abstract graphs are materialized such that the left hand side of a given rule is concretely present in the materialization. Then, second, the rule is applied the concrete subgraph and third the resulting graph is normalized using neighborhood abstraction.

In [19] a shape analysis of programs is proposed. The abstract domain used in this approach is the same as ours except that they use instrumentation predicates instead of shape constraints. Similar as in the work above and similar to our approach, materialization is utilized to improve the precision of abstract transformations. However, to yield precise abstract transformers, the approach based on instrumentation predicates requires for *hand-written* update formulae for each instrumentation predicate and each transformation rule. In [16] the authors improve these results by using finite differencing to *automatically* determine update formulae for the instrumentation predicates, which yet might be less precise than hand-written ones. Another, more recent approach for the computation of best abstract transformers in the domain is presented in [23]. There, the idea is to characterize shape graphs by first-order logic formulae [24] and then use decisions procedures, similar to [17], to compute the best abstract transformer.

Future Work. In the future, we want to look at finite subsets of our abstract domain that allow us to guarantee the termination of the abstract exploration. When doing so, on the one hand, the precision of the finite subsets should be still tunable such that, in principle, a future CEGAR (counter-example guided abstraction refinement, [12]) approach can be employed. On the other hand, the finite subsets should be defined such that the best abstract transformer is still efficiently computable.

References

1. Baldan, P., Corradini, A., König, B.: A framework for the verification of infinite-state graph transformation systems. Information and Computation 206(7), 869–907 (2008)
2. Ball, T., Podelski, A., Rajamani, S.: Boolean and cartesian abstraction for model checking c programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)

3. Bauer, J., Wilhelm, R.: Static analysis of dynamic communication systems by partner abstraction. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 249–264. Springer, Heidelberg (2007)
4. Bauer, J., Boneva, I., Kurbán, M., Rensink, A.: A modal-logic based graph abstraction. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 321–335. Springer, Heidelberg (2008)
5. Bisztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings by rule extraction. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 347–361. Springer, Heidelberg (2008)
6. Bogudlov, I., Lev-Ami, T., Reps, T., Sagiv, M.: Revamping TVLA: making parametric shape analysis competitive. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 221–225. Springer, Heidelberg (2007)
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1979, pp. 269–282. ACM, New York (1979)
8. Engels, G., Güldali, B., Soltenborn, C., Wehrheim, H.: Assuring consistency of business process models and web services using visual contracts. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 17–31. Springer, Heidelberg (2008)
9. Fitting, M.: Kleene's three valued logics and their children. FI 20, 113–131 (1994)
10. Graf, S., Saidi, H.: Construction of abstract state graphs with pvs. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
11. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: Showing full semantics preservation in model transformation - a comparison of techniques. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 183–198. Springer, Heidelberg (2010)
12. Jha, S., Lu, Y., Grumberg, O., Clarke, E., Veith, H.: Counterexample-guided Abstraction Refinement. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 152–165. Springer, Heidelberg (2006)
13. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science 109(1-2), 181–224 (1993)
14. Rensink, A.: The GROOVE simulator: A tool for state space generation. Applications of Graph Transformations with Industrial Relevance, 479–485 (2004)
15. Rensink, A., Distefano, D.: Abstract graph transformation. Electr. Notes Theor. Comput. Sci. 157(1), 39–59 (2006)
16. Reps, T., Sagiv, M., Loginov, A.: Finite differencing of logical formulas for static analysis. ACM Trans. Program. Lang. Syst. 32, 24:1–24:55 (2010)
17. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 3–25. Springer, Heidelberg (2004)
18. Reps, T.W., Sagiv, S., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004)
19. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
20. Saksena, M., Wibling, O., Jonsson, B.: Graph grammar modeling and verification of ad hoc routing protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 18–32. Springer, Heidelberg (2008)

21. Steenken, D., Wehrheim, H., Wonisch, D.: Towards a shape analysis for graph transformation systems. In: Proceedings of the 22nd Nordic Workshop on Programming Theory (2010), Technical Report, http://www.cs.uni-paderborn.de/fileadmin/Informatik/AG-Wehrheim/Personen/Dominik_Steenken/ShapeAnalysis2010TR.pdf
22. Wonisch, D.: Increasing the preciseness of shape analysis for graph transformation systems. Master's thesis, University of Paderborn (August 2010)
23. Yorsh, G., Reps, T., Sagiv, M.: Symbolically computing most-precise abstract operations for shape analysis. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 530–545. Springer, Heidelberg (2004)
24. Yorsh, G., Reps, T., Sagiv, M., Wilhelm, R.: Logical characterizations of heap abstractions. ACM Trans. Comput. Logic 8 (January 2007)

On the Specification, Verification and Implementation of Model Transformations with Transformation Contracts

Christiano Braga, Roberto Menezes, Thiago Comicio,
Cassio Santos, and Edson Landim

Instituto de Computação, Universidade Federal Fluminense, Brazil
{cbraga,rmenzes,tcomicio,cfernando,elandim}@ic.uff.br

Abstract. Model transformations are first-class artifacts in a model-driven development process. As such, their verification and validation is an important task. We have been developing a technique to specify, verify, validate and implement model transformations. Our technique is based on the concept of *transformation contracts*, a specification that relates two modeling languages and declares properties that must be fulfilled in such a relation. A transformation contract is essentially a transformation model that allows for the verification and validation of a model transformation using the same techniques one uses to verify and validate any given model. This paper describes our technique, discusses *consistency* of model transformations and reports on its application to a model transformation from access control models to Java security.

1 Introduction

Model-driven development (MDD, e.g. [15]) is a software engineering discipline that considers models as *live artifacts* in the development process. By live artifacts we mean that models are not used for documentation purposes only but actually as input to software tools that may operate on them and produce other artifacts. Such artifacts may be compilable source-code or other models, in the same or different abstraction levels than the source model. MDD aims at allowing for a *generative* software development process in which applications are produced out of models possibly described at the application domain level.

Model transformations are first-class artifacts in a model-driven development process. As such, their specification, verification and validation are important tasks in an MDD process. A transformation contract [10, 14, 7, 8, 12] is a specification of a model transformation. Essentially, a transformation contract is comprised by *relations* between the model elements of the modeling languages it relates and *properties* that such relations must fulfill. Therefore, a model transformation specification may be understood as a metamodel. In this paper, we follow the terminology of [6] and call the metamodel representing a model transformation a *transformation metamodel*. A particular application of a model transformation is represented as an object model instance of the transformation metamodel. A transformation contract is thus a transformation metamodel and a set of properties over it. Under this perspective, *model reasoning* techniques may be applied to *reason about model transformations* as well.

More formally, we say that a model m is *well-formed* with respect to a metamodel M, denoted by $m \in M$, if m is syntactically a proper instance of M, that is, essentially, the objects in m are instances of metaclasses in M and links in m are instances of associations in M. A model m is *in conformance* with M, denoted by $m \models P_M$, if the properties of the metamodel M hold in m. A transformation contract (see Section 4) between two modeling languages M and M' related by a set of associations A, denoted by $M \bowtie_A M'$, is a *pair* comprised by the *transformation metamodel* resulting from the disjoint union of M, M' and A, and *a set of properties* over the transformation metamodel. A model transformation is said *correct* with respect to a transformation contract $M \bowtie_A M'$ iff $m \models P_M \Rightarrow ((m' \models P'_M) \wedge ((m \bowtie_l m') \models P_{M \bowtie_A M'}))$ *if* $m \in M, m' \in M', l \in A, m \bowtie_l m' \in M \bowtie_A M'$, where P_M are the properties of the metamodel M and $m \bowtie_l m'$ is an instance of the transformation metamodel of $M \bowtie_A M'$ with l a set of links well-formed with respect to A.

In [10,14,7,8], the authors specify such properties as invariants in the Object Constraint Language representing *typing rules* of the different metamodels involved in a model transformation. In this paper we move forward by generalizing previous work by allowing the automatic verification of *model transformation consistency* understood as satisfiability of an associated theory in Description Logic [2]. We also discuss the implementation of transformation contracts as the application of a *design pattern* that captures our way of designing transformation contracts in the context of a model transformation from access control models to Java security. Space constraints prevent us from discussing other applications of our approach. We refer the interested reader to `http://lse.ic.uff.br` for a model transformation from UML class diagrams to Enterprise Java Beans code and other tools.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes our proposed model transformation process. Section 4 gives an algebraic definition of transformation contracts and exemplifies it in the context of our example model transformation from access control models to Java security. Section 5 describes how consistency verification may be added to our model transformation process and exemplifies its application. Section 6 reports on a design pattern for the implementation of model transformations according to our proposed model transformation process and exemplifies its application. We conclude this paper in Section 7 with our final remarks.

2 Related Work

Our previous work in [7,8,12] agrees in spirit with [10,14]. However, there are differences at the specification, verification and implementation levels. At the specification level, we adopt a *relational* approach towards the specification of a model transformation, similar in essence to [1,6] but different from [10,14] where transformation contracts are specified as OCL invariants from source to target model elements. In [1,6,14] their approach is discussed informally while in Section 4 transformation contracts are formalized as algebras. The specification of a relation between the model elements of the metamodels related by a model transformation is essential to generalize from OCL invariants and understand that different *kinds* of properties may be specified *over such relation*. It is important to make *explicit* the relationship among the metamodels.

At the verification level, different kinds of properties may be reasoned upon despite OCL invariants. One such property is model consistency understood as satisfiability of the description logic theory (or knowledge base or TBox, in description logic terminology) associated with a given model. Note that given the perspective that we apply here, model transformations may also be checked for consistency, as it is also a metamodel! At the implementation level, given our generalization, we are not confined to OCL based languages such as OMG's Query View Transformation [16] to specify our transformation contracts. QVT is one possibility, that allows for the specification of the relation among the metamodels related by the model transformation. However, it should be clear that it is one particular implementation of the the transformation metamodel, not the only one. Moreover, we also generalize the understanding of concrete and abstract syntaxes discussed in [1, 14]. There, the authors understand that the concrete syntax of a modeling language must be in bijection with the abstract syntax described by the metamodel. This has the benefit for the model transformation designer to have a parser for any given modeling language. However, this choice is cumbersome for the user since one must create a quite detailed model so the machine may understand what one wants. The idea of a *domain-specific* modeling language is precisely to allow descriptions at the domain level and if possible concise ones. This is what UML profiles are for. In the design pattern described in Section 6, we allow the model transformation designer (actually the modeling language designer) to define how a model must be represented as an instance of a metamodel and its inverse, in the form of *parsing* and *pretty-printing* functions.

QVT and Triple Graph Grammars [17] (TGG) are other possible specification frameworks for a model transformation that requires specific theory and machinery to reason and implement model transformations. The transformation contracts approach proposed in this paper is not biased to any such specification languages and may be used with them as well. Note, however, that QVT is bound to OCL as the specification language for the properties of metamodels and there are properties best specified in other semantic frameworks, such as consistency in DL as discussed in this paper. TGG may not have QVT's restriction on the specification language for metamodel properties but another aspect of our approach is that we apply to model transformation specification design the same specification languages and techniques one would to design a modeling language. No additional framework, such as QVT or TGG, is necessary.

In [6] the authors discuss the idea of a transformation models to specify a model transformation. We share the idea of transformation models but in this work is make precise what we mean by transformation model and how it may be used to reason on model transformations.

3 Model-Driven Development with Transformation Contracts

Model transformations relate *languages*. If one decides to work with the standards of the Object Management Group (OMG) to take advantage of the interoperability gained from using such standards, the abstract syntax of the languages related by a model transformation may be described in the form of a UML class diagram. Such a model is called a *metamodel* since it describes the syntax that models should follow.

A model that is a proper instance of a metamodel is called *well-formed* with respect to the metamodel. The notion of well-formedness may be understood as the pertinence of a program with respect to the programming language it is written in, that is, a model must be well-formed with respect to its metamodel as a program written in a language L must be well-formed with respect to L's syntax. For example, UML has a metamodel and any UML class diagram may be seen as an instance of the UML metamodel that should be well-formed with respect to it. Figure 1 shows a simplified version of the UML metamodel, slightly enhanced from [15] by considering inheritance between classes through the association *inherited-inheritsFrom*. The metamodel essentially represents the notions of datatypes, classes, attributes, operations, interfaces, association ends, their inter-relations and their typing relations.

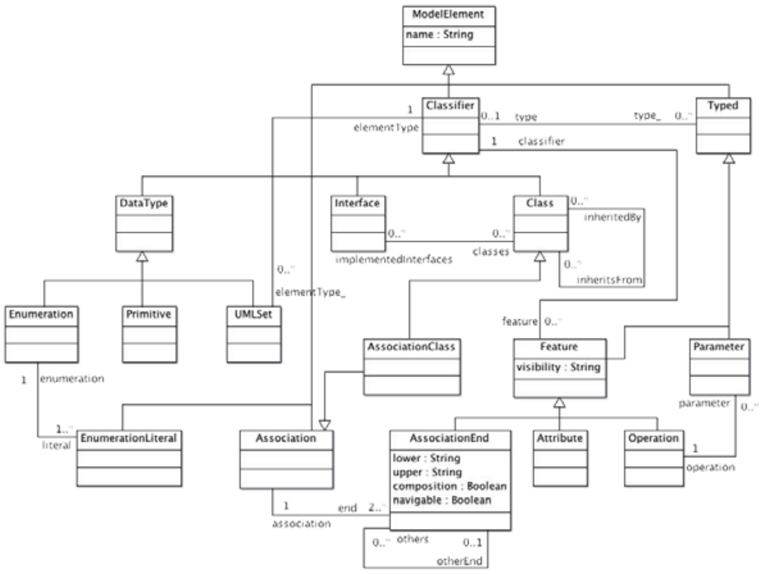


Fig. 1. Simplified UML metamodel

UML is an extensible modeling language. Any given UML model element may be *tagged* or stereotyped, using UML terminology, in order to denote a new entity named after the tag's name. This is "UML's way" of defining *domain specific modeling languages*. We may now, once again, draw a relationship between model-driven concepts and programming languages concepts. One may understand a UML profile, which is essentially a UML extension consisting of a set of stereotypes and other extension elements, as the *concrete syntax* of a modeling language M. The metamodel of M may be understood as its *abstract syntax*. With that understanding in mind, the first step that a model transformation should do when transforming an UML model m written using an UML profile that represents a modeling language M is to map m into an instance of M's metamodel. This step is similar to language *parsing* in a compiler.

Up to this point, the model-driven development process we are describing in this paper can be drawn as follows, where m, m', n, n' are models; $\mathcal{S}$ and $\mathcal{T}$ represent the source and target metamodels related by a model transformation τ; we write $m \in M$ to denote that the model m is well-formed with respect to M where M represents the concrete syntax for a modeling language M and $\mathcal{M}$ represents the abstract syntax of the modeling language M; $parse$ is the mapping that given a model m written in the concrete syntax of a modeling language M (S and T in the diagram) generates an abstract syntax version m' of m where m' is well-formed with respect to $\mathcal{M}$; finally, $pretty\ print$ is the inverse mapping of $parse$, that is, it generates the concrete syntax of the modeling language M given a model instance of $\mathcal{M}$,

$$m \in S \xrightarrow{parse} m' \in \mathcal{S} \xrightarrow{\tau} n \in \mathcal{T} \xrightarrow{pretty\ print} n' \in T.$$

It is not always true, however, that any well-formed model with respect to a given metamodel is *in conformance* with it. For instance, a UML class model m with an inheritance chain that has a cycle may be syntactically well-formed with respect to UML's metamodel but it is not in conformance with it. The reason is that there is an *invariant* in the UML metamodel that specifies that there should be no cycles in any inheritance chain. Since the invariant does not hold in m, the model m is not in conformance with UML's metamodel. The conformance relation between a model m and a metamodel M is given by well-formedness of m with respect to M and validity of the invariants of M in m, assuming M consistent, that is, assuming that M has instances. The conformance relation between a metamodel and an instance of it is similar to the concept of *type checking* in programming languages. A syntactically correct program p with respect to a language L is ill-typed if the typing rules of L do not apply to p.

One way to specify such invariants is using the Object Constraint Language (OCL). Essentially, OCL has several constructs for manipulating collections of typed model elements in a model m, navigating through m's relationships, defining operations and invariants in M, where M is the metamodel of m. For example, the invariant *noCyclesinClassHierarchy* below checks for the presence of cycles in class hierarchies in a model instance of the UML metamodel by verifying for each class c if c is not included in the transitive closure of the *inheritsFrom* relationship that represents class inheritance hierarchy. The invariant uses two operations, namely *superPlus* and *superPlusOnSet*, to calculate the transitive closure. The operation *superPlusOnSet* does the actual calculation by a recursive call on each element of the collection yielded by the *inheritsFrom* relation for each class c. Regarding OCL syntax, the keyword *context* defines the type of objects that the invariant should be applied to. The keyword *inv* defines an invariant. The informal meaning of the remaining OCL constructors in the example are as follows: *forAll* iterates over the elements of a given collection checking for a given predicate; *excludes* checks if a given collection does not contain a given element; *collects* creates a collection of objects such that a given predicate holds; *flatten* receives a set which may have other sets as elements and produces a flatten set of objects from its set elements; *asSet* casts a collection into a set; and *including* includes a given element in a given collection. The user-defined function *emptySet* constructs an empty set of objects of type *Class*.

```
context Class inv noCyclesinClassHierarchy: self.inheritsFrom→forAll(r|r.superPlus()→excludes(self))
context Class::superPlus():Set(Class) body: self.superPlusOnSet(self.emptySet())
context Class::superPlusOnSet(rs:Set(Class)):Set(Class) body:
if self.inheritsFrom→notEmpty() and rs→excludes(self)
then self.inheritsFrom→collect(c : Class | c.superPlusOnSet(rs→including(self)))→flatten()→asSet()
else rs→including(self) endif
```

OCL can be used to *automatically* validate UML models. Considering an implementation of an OCL interpreter, such as [11], one may actually apply the invariants of a metamodel M to a syntactically well-formed model m with respect to M to guarantee m's conformance with respect to M. Therefore, before applying a model transformation to a given model m, one must make sure that m is syntactically well-formed with respect to M and all invariants in M (such as *noCyclesinClassHierarchy*) hold in m. For example, a UML class diagram must be well-formed with respect to the metamodel in Figure 1 and the invariant *noCyclesinClassHierarchy* should hold on it.

The MDD process adopted in this paper when invariants are considered may be drawn as follows where $I_{\mathcal{M}}$ are the invariants of the metamodel of the modeling language M and $m \models I_{\mathcal{M}}$ means that all the invariants in $I_{\mathcal{M}}$ hold in the model $m \in \mathcal{M}$,

$$m \in S \xrightarrow{parse} m' \in \mathcal{S}, m' \models I_{\mathcal{S}} \xrightarrow{\tau} n \in \mathcal{T}, n \models I_{\mathcal{T}} \xrightarrow{pretty\ print} n' \in T.$$

A transformation contract is a specification of *what* a model transformation should do. It is written in the form of invariants that must hold in the transformation metamodel of the source and target languages related by a set of associations. By transformation metamodel we mean a metamodel $\mathcal{K}$ resulting from a model operation $\mathcal{S} \bowtie_{A_{\mathcal{K}}} \mathcal{T}$ on two given metamodels $\mathcal{S}$ and $\mathcal{T}$ that *extends* the metamodels $\mathcal{S}$ and $\mathcal{T}$ by: (i) disjointly uniting all the model elements of $\mathcal{S}$ and $\mathcal{T}$; (ii) declaring associations $a \in A_{\mathcal{K}}$ that relate classes in $\mathcal{S}$ with $\mathcal{T}$ and disjointly uniting $A_{\mathcal{K}}$ with $\mathcal{S}$ and $\mathcal{T}$; and (iii) declaring invariants $I_{\mathcal{K}}$ over $A_{\mathcal{K}}$. The MDD process adopted in this paper when transformation contracts are considered may be drawn as follows where $\mathcal{K} = \mathcal{S} \bowtie_{A_{\mathcal{K}}} \mathcal{T}$, $k \in \mathcal{K}$, $l \in A_{\mathcal{K}}$ and $k = (m \bowtie_l n)$,

$$m \in S \xrightarrow{parse} m' \in \mathcal{S}, m' \models I_{\mathcal{S}} \xrightarrow{\tau} n \in \mathcal{T}, n \models I_{\mathcal{T}}, k \models I_{\mathcal{K}} \xrightarrow{pretty\ print} n' \in T.$$

4 Specifying Transformation Contracts

In this section we formalize algebraically the concept of transformation contracts and exemplify its specification. We begin with the formal definitions and then, for our example, we represent metamodels as UML class diagrams constrained by expressions in OCL. The equational interpretation of a class diagram means essentially to understand it as an algebraic signature where a class declaration is formalized as a sort declaration with an appropriate constructor operation and an association declaration is formalized as an operation over the appropriate sorts representing the classes that the given association relates. Cardinality constrains and OCL invariants are formalized as equations over the signature defined from class and association declarations. For OCL in particular, there is a general theory for basic OCL operations which is extended (in a precise algebraic sense) for each OCL constrained class diagram. (We refer the interested reader to [13] for an algebraic formalization of OCL.) The formal definitions below are used in the example, described later in this section, to make it precise.

Definition 1 (Equational theory). *An equational theory $\mathcal{M}$ is a structure $\langle \Sigma, E \rangle$ where Σ is the signature of $\mathcal{M}$ and E is a set of terminating and confluent equations over Σ.*

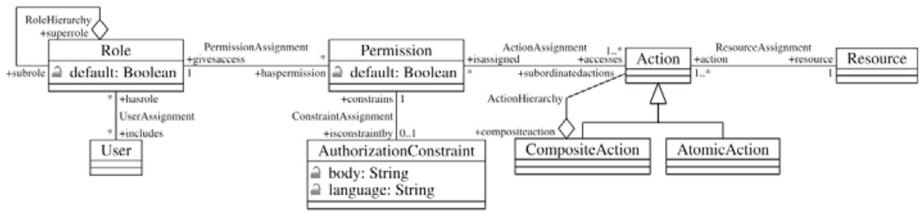
Definition 2 (Metamodel). *A metamodel $\mathcal{M}$ of a modeling language M is an equational theory $\mathcal{M} = \langle C \cup A, I \rangle$ where C is the signature defining the metaclasses of M, A is the signature defining the associations of M, and I is a set of equations over $C \cup A$ representing the invariants of M.*

Definition 3 (Transformation contract). *A transformation contract $S \bowtie_A T$ between modeling languages S and T related by the associations in A is an equational theory $\mathcal{K} = \langle (C_\mathcal{S} \cup A_\mathcal{S}) \uplus (C_\mathcal{T} \cup A_\mathcal{T}) \uplus A_\mathcal{K}, I_\mathcal{S} \cup I_\mathcal{T} \cup I_\mathcal{K} \rangle$, where $\uplus$ is the disjoint union operation over sets, $\mathcal{S} = \langle (C_\mathcal{S} \cup A_\mathcal{S}), I_\mathcal{S} \rangle$ is the metamodel of the modeling language S, $\mathcal{T} = \langle (C_\mathcal{T} \cup A_\mathcal{T}), I_\mathcal{T} \rangle$ is the metamodel of the modeling language T, $A_\mathcal{K}$ is a signature representing associations in A, and $I_\mathcal{K}$ is a set of equations over $A_\mathcal{K}$ representing invariants over the associations between S and T.*

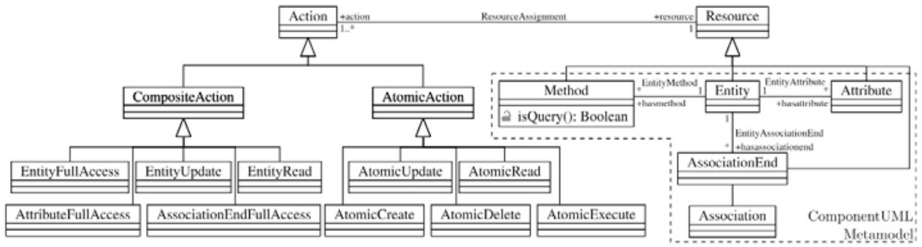
We exemplify the specification of a transformation contract with an excerpt, due to space constraints, of the model transformation from the platform independent modeling language SecureUML+ComponentUML [4], for access control modeling, to a platform specific modeling language we call JAAS that represents the Java Authentication and Authorization Service. This excerpt is part of the SecureUMLtoJAAS+AAC model transformer that generates AspectJ code, an extension of the Java programming language with aspect-oriented concepts, with JAAS support from access control models for Java-based applications. The tool is available for download from `http://lse.ic.uff.br`. The complete description of the model transformation is given in [12].

The modeling languages. SecureUML+ComponetUML is a language to model access control. A SecureUML+ComponentUML model describes permissions that user roles have in order to perform actions over entities. Examples of such actions are: (i) the *execution* of a method, (ii) *updating* an entity's state, or (iii) *full access* to an entity. The first two actions are *atomic actions* and the last one a *composite action*. As opposed to atomic actions, composite actions form a collection of actions which may be atomic or composite. The *EntityFullAccess* composite action, for instance, allows for both read and update access to all elements of an entity, that is, its attributes, methods and association ends. It includes *EntityRead* and *EntityUpdate* which in turn include *AtomicRead* and *AtomicUpdate*, respectively. SecureUML also allows for the modeling of user roles' *hierarchies*. Role inheritance means that if role r_1 inherits from role r_2 than all permissions of r_2 also apply to r_1. The metamodel of SecureUML+ComponentUML is given in Figure 2.

We have defined a modeling language called JAAS, which is the acronym for the Java Authorization and Authentication Service, to capture the access control subset of the Java security framework. Its metamodel is depicted in the diagram in Figure 3.



(a) SecureUML metamodel



(b) ComponentUML metamodel

Fig. 2. SecureUML+ComponentUML metamodel

In JAAS there are different authentication mechanisms such as LDAP or NIS. These mechanisms are captured as instances of a protection domain. The metaclasses *Principal*, *JAASPermission* and *JAASAction* are the counter parts of *Role*, *Permission* and *Action* in SecureUML. We will focus on the transformation from *Role*, *Permission* and *Action* to *Principal*, *JAASPermission* and *JAASAction* in this paper.

The transformation contract. The "raison d'être" of a transformation contract is to guarantee that essential properties of the source model are preserved in the target model. In our example, we want to guarantee that a user in a given SecureUML role is properly represented as a principal, that is, a principal may enact the same actions, with the same constraints, of its associated role, no more no less.

As opposed to SecureUML, JAAS does not have role hierarchies or composite action hierarchies. The transformation contract from SecureUML+ComponentUML model to JAAS models is the result of a *composition* of two contracts: (i) the *flattening* contract $\mathcal{F}$, in which the role and action hierarchies are flattened in SecureUML and (ii) the *mapping* contract $\mathcal{M}$, in which flattened SecureUML and JAAS are related. With the composed contract, a principal will be able to enact the actions associated with the permissions of the role that the given principal is related with, since: (i) flattening the role hierarchy associates with a given role all the permissions of the transitive closure of its inheritance hierarchy and (ii) flattening the action hierarchy associates with a given role all the atomic actions, and their constraints, for each composite action in a given permission.

Recall from Definition 3 that a contract is a structure $\mathcal{K} = \langle (C_{\mathcal{S}} \cup A_{\mathcal{S}}) \uplus (C_{\mathcal{T}} \cup A_{\mathcal{T}}) \uplus A_{\mathcal{K}}, I_{\mathcal{S}} \cup I_{\mathcal{T}} \cup I_{\mathcal{K}} \rangle$. For the flattening contract $\mathcal{F}$, $C_{\mathcal{S}}$ and $A_{\mathcal{S}}$ are the metaclasses and associations from the SecureUML+ComponentUML metamodel. The invariants

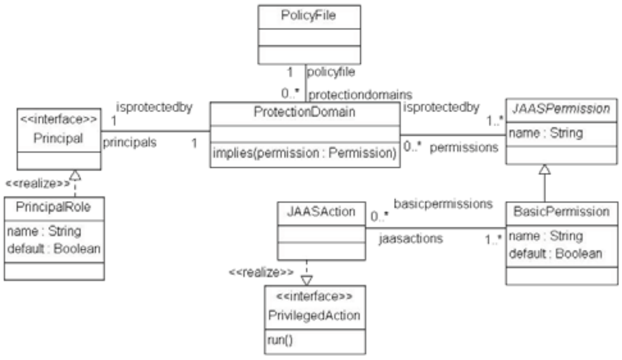


Fig. 3. JAAS metamodel

$I_{\mathcal{S}}$ will *not* be discussed here as they are not relevant to our example. (For the sake of exemplification, one such invariant is the need of a *default role* that every role must inherit from and that has a *default permission* over resources, in any given SecureUML model.) The set $C_{\mathcal{T}}$ includes metaclasses *FRole* and *FAction*, for flattened role and flattened action, respectively. The set $A_{\mathcal{T}}$ includes[1]: (i) a one-to-one association *role-frole* between metaclasses *Role* in SecureUML and *FRole* in flattened SecureUML, (ii) a one-to-one association *atomicaction-faction* between *AtomicAction* in SecureUML and *FAction* in flattened SecureUML, (iii) a one-to-many association *frole-permission* between *FRole* in flattened SecureUML and *Permission* in SecureUML, and finally, (iv) a one-to-many association *faction-permission* between *FAction* in flattened SecureUML and *Permission* in SecureUML. The set $I_{\mathcal{T}}$ is empty since $\mathcal{F}$ is an endogenous transformation in SecureUML that substitutes the role and action hierarchies for equivalent ones without inheritance. Therefore, there are no invariants in the target of $\mathcal{F}$ since the contract is only about flattening.

The set $I_{\mathcal{K}}$ is the interesting one as it specifies the flattening process. The first invariant in $I_{\mathcal{K}}$, roleFlattening, specifies that an *FRole* "mirror" instance of a *Role* in SecureUML model has the same permissions of the reflexive-transitive closure of the superrole relation of the given *Role*. The operation allPermissions is defined in [3] and calculates all the *Permissions* of the transitive closure of the *superrole* relation between instances of *Role* in SecureUML.

```
context FRole inv roleFlattening:
self.frole-permission→includesAll(self.role-frole→allPermissions())
context Role::allPermissions():Set(Permission) body: self.superrolePlus().haspermission→asSet()
context Role::superrolePlus():Set(Role) body: self.superrolePlusOnSet(self.superrole)
context Role::superrolePlusOnSet(rs:Set(Role)):Set(Role) body:
if rs.superrole→exists(r|rs→excludes(r))
then self.superrolePlusOnSet(rs→union(rs.superrole)→asSet())
else rs→including(self) endif
```

[1] There could be associations between a class in the source metamodel and different classes in the target metamodel. This example is functional but it should be clear that $A_{\mathcal{T}}$ denotes a relation.

The invariant actionFlattening specifies that every *FAction* instance has the same permissions as its *AtomicAction* counterpart which means gathering the permissions of all *CompositeAction* that the given *AtomicAction* is part of together with the permissions attached to the *AtomicAction* itself. The operation allPermissions for *AtomicAction* is calculated in a way similar to *Role* but using the transitive closure of the *compositeactions* relation.

```
context FAction inv actionFlattening:
self.faction-permissions→includesAll(self.atomicaction-faction→allPermissions())
```

Once the hierarchies are flattened, the mapping contract $\mathcal{M}$ from flattened SecureUML to JAAS, with respect to *FRole*, *Permission*, *FAction*, *PrincipalRole* and *JAASAction*, is trivial. The signatures in $\mathcal{M}$ are given by the metaclasses and associations of flattened SecureUML and JAAS. The set $I_{\mathcal{K}}$ of $\mathcal{M}$ essentially establishes a bijection between *FRole* and *PrincipalRole* and a bijection between *FAction* and *JAASAction*: for every *FRole* there must exist a *PrincipalRole* such that the *PrincipalRole*'s associated instances of *JAASAction* are those related with the instances of *Permission* of the given *FRole*. There are other aspects of model transformation that are handled by $\mathcal{M}$ but are out of the scope of this paper.

5 Verifying and Validating Transformation Contracts

5.1 Model Consistency Reasoning and Description Logic

In Section 3 we have outlined our model-driven development process with transformation contracts. We defined that a model m is in *conformance* with its metamodel $\mathcal{M}$ if $m \models I_{\mathcal{M}}$, that is, if $I_{\mathcal{M}}$, the invariants of $\mathcal{M}$, hold in m. This definition is sound under the assumption that both $\mathcal{M}$ and m are *consistent*, that is, that they may be *instantiated*. An example of inconsistency is as follows: assume that inheritance is a *complete* and *disjoint* relation, that is, if classes B and C inherit from A then A is completely defined by the union of B and C and that B and C are disjoint sets. Now consider that, perhaps after a refactoring operation in a model, B also inherits from C. Clearly, this is an inconsistent model as B can not be included in C and disjoint with C at the same time.

Description logic [2] is a family of logics defined to be efficiently decidable. Each fragment of the logic was carefully studied on its expressiveness and efficiency of reasoning. Consistency reasoning is a decision procedure commonly associated with DL reasoning. DL consistency reasoning may be applied to class diagrams when a proper encoding is defined between class diagrams and DL theories (or knowledge bases, in DL terminology). Such a mapping has been defined in [5], proven correct and the complexity of DL reasoning on class diagrams has been calculated. The encoding of class diagrams in DL essentially relates classes with DL concepts, which denote sets, and associations with DL roles, which are binary relations. Class diagrams are encoded in the logic $\mathcal{ALCQI}$ which is a DL that allows for the specification of: (i) cardinality constraints over roles, denoted by axioms of the general form $\geq n\ R.C$ where n is a natural number, R is a role and C is a concept, that constrain the number of individuals (or instances) of C to be at least n in the relation R, (ii) concept negation, denoted by formulas of the form $\neg C$ where C is a concept, specifying the set of individuals that do not belong to the set denoted by C, (iii) concept conjunction, denoted by formulas of the form $C_1 \sqcap C_2$ where C_1 and C_2 are concepts, which specifies union of the sets denoted by C_1 and C_2 and (iv) definition of inverse of roles, denoted by formulas of the form R^- where R is a role, specifying the inverse relation of R.

For the purposes of this paper, it suffices to explain the encoding for classes, inheritance and binary associations. In [5, Section 7.1] they are described as follows: (i) A *class* C is represented by an atomic concept C; (ii) A *generalization* between a class C and its child class C_1 can be represented using the inclusion assertion $C_1 \sqsubseteq C$. A class hierarchy can be represented by the assertions $C_1 \sqsubseteq C, \ldots, C_n \sqsubseteq C$ when C_i inherits from C. A disjointness constraint among classes $C_1, \ldots, C_n$ can be modeled as $C_i \sqsubseteq \bigsqcap_{j=i+1}^{n} \neg C_j$, with $1 \leq i \leq n-1$, while a covering constraint can be expressed as $C \sqsubseteq \bigsqcup_{i=1}^{n} C_i$; (iii) Each binary association (or aggregation) A between a class C_1 and a class C_2, with multiplicities $m_l..m_u$ and $n_l..n_u$ on each end, respectively, is represented by the atomic role A, together with the inclusion assertion $\top \sqsubseteq \forall A.C_2 \sqcap \forall A^-.C_1$. The multiplicities are formalized by the assertions $C_1 \sqsubseteq (\geq n_l\ A.\top) \sqcap (\leq n_u\ A.\top)$ and $C_2 \sqsubseteq (\geq m_l A^-.\top) \sqcap (\leq m_u A^-.\top)$, where $\top$ denotes the largest concept (top) that includes all concepts and $\forall R.C$ is just syntactic sugar for $\leq 0\ R.\neg C$.

5.2 Model Consistency Verification in Model Transformations with Transformation Contracts

We have incorporated consistency reasoning into our MDD process with transformation contracts. The idea is essentially to check for model consistency before validating the appropriate invariants as it only makes sense to check for invariants of models that are consistent. Concept inconsistency is denoted by $C \sqsubseteq \bot$, where C is a concept and $\bot$ denotes the empty (bottom) concept. A model m is consistent iff $\forall C \in \mathit{classesOf}(m).\neg(C \sqsubseteq \bot)$ where $\mathit{classesOf}(m)$ denotes the set of concepts that encode classes of a model m. When model consistency is considered, our model transformation process may be drawn as follows, where $\mathcal{K} = \mathcal{S} \bowtie \mathcal{T}$, $k \in \mathcal{K}$, and $k = (m \bowtie n)$. Note that checking for consistency of models n and k *is* necessary as the new relations introduced in k may turn classes both in m' and in n inconsistent.

$$\begin{tikzcd}
m \in S \arrow[rr, "parse"] & & \begin{array}{c} m' \in \mathcal{S}, m' \models I_{\mathcal{S}}, \\ (\forall C \in \mathit{classesOf}(m').\neg(C \sqsubseteq \bot)) \end{array} \arrow[dll, "\mathcal{T}"'] \\
\begin{array}{c} n \in \mathcal{T}, n \models I_{\mathcal{T}}, k \models I_{\mathcal{K}}, \\ (\forall C \in \mathit{classesOf}(n).\neg(C \sqsubseteq \bot)), \\ (\forall C \in \mathit{classesOf}(k).\neg(C \sqsubseteq \bot)) \end{array} \arrow[rr, "pretty\ print"] & & n' \in T
\end{tikzcd}$$

To verify the consistency of a model $m \in \mathcal{M}$ it is necessary, of course, to define an encoding of $\mathcal{M}$ in an description logic, such as the authors did for UML class diagrams into $\mathcal{ALCQI}$ in [5]. The encoding will depend on the reasoning procedure that will be used. There are two types of reasoning procedures in DL. The so-called ABox reasoning means to check that the axioms of a knowledge base (also called TBox) hold on a particular set of individuals (or instances) of concepts and roles. The so-called TBox analysis means to perform a general symbolic reasoning process over a given TBox that verifies if the axioms of a TBox are generally satisfiable (and not only for a particular set of individuals).

The ABox analysis of a model m instance of a metamodel $\mathcal{M}$ requires the representation of the metamodel $\mathcal{M}$ as a TBox and m as an ABox, that is, a set of individuals. The ABox analysis process consists of checking that the axioms of the TBox representation of $\mathcal{M}$ hold in the ABox representation of m. The TBox analysis of a model m requires an *extension* of the TBox that represents the metamodel $\mathcal{M}$ of m with concepts and roles representing the classes and associations of $\mathcal{M}$, following the encoding defined in 5.1. Essentially, the axioms representing m are defined as follows, assuming m well-formed with respect to $\mathcal{M}$: (i) the concepts that represent metaclasses in $\mathcal{M}$ are subsumed by concepts representing objects, of the appropriate class, in m, (ii) roles representing associations in $\mathcal{M}$ are subsumed by roles representing links, of the appropriate associations, between objects in m, and (iii) include axioms to constrain roles representing links in m, following the encoding in Section 5.1. We have chosen TBox analysis since it is more general than ABox analysis.

As a concluding remark for this section, let us discuss a bit on the combination of consistency verification in DL and invariant validation in OCL. It is out of the scope of this paper, however, a detailed discussion on this subject as the objective of this paper is to discuss how model transformations may be developed rigorously with transformation contracts. It should be clear that consistency verification in DL is *general*. Using DL one may check for: (i) *metamodel consistency*, in other words, answer the question "Does this modeling language admit models?", and therefore reason about the consistency of a modeling language, and (ii) *model consistency*, in other words, answer the question "Does this model admit scenarios?" and therefore reason about the possibility of the instantiation of a particular, that is, the existence of scenarios for a given model. Note that to answer these questions in general we need TBox reasoning. OCL validation executes of OCL invariants on a particular scenario or metamodel instance. It does not allow any reasoning at the modeling language level. However, both techniques allow for reasoning of models. At this point one may wonder why OCL validation is necessary at all if DL reasoning is considered. The answer is that they are *complementary techniques*, as pointed out in [9], in the sense that OCL validation, that is, the execution of OCL invariants over models, identifies errors that DL reasoning may miss. DL has the so-called *open world assumption* which means that a *missing* link between objects, for instance, is not considered an error, as opposed to the so-called *closed world assumption*, where the absence of information, such as a missing link between objects for example, is an error. Therefore, if DL reasoning does not identify a problem because there is missing information in a model then the execution of OCL invariants would. This is the reason why these techniques should be applied sequentially starting with DL reasoning.

5.3 Verifying and Validating Access Control Models

As an illustrative example, let us consider the consistency analysis of access control models in SecureUML+ComponentUML. We discuss two scenarios: (i) DL reasoning identifies a problem, and (ii) DL reasoning does not identify a problem due to the open world assumption but OCL validation does.

For the first scenario, let us consider a model m, instance of the SecureUML+ComponentUML in Figure 2, that contains an authorization constraint a associated with two permissions p_1 and p_2 through its *ConstraintAssignment* association. The knowledge base of m extends the knowledge base that represents the metamodel of SecureUML+ComponentUML with axioms representing objects and links using the metamodel and model representations required by TBox analysis, described in Section 5.2. An excerpt of the knowledge base, for model elements a, p_1 and p_2 and the concepts they extend, is as follows: (i) from the knowledge base for SecureUML+ComponentUML metamodel,

$$\begin{aligned}
\top &\sqsubseteq \forall ConstraintAssignment^{-}.Permission \sqcap && (1)\\
&\quad \forall ConstraintAssignment.AuthorizationConstraint && \\
AuthorizationConstraint &\sqsubseteq (\equiv 1\ ConstraintAssignment.\top), && (2)\\
&\quad \text{where}(\equiv 1 C.R) \text{ is syntactic sugar for} && \\
&\quad \geq 1 C.R \sqcap \leq 1 C.R, C \text{ is a concept and } R \text{ is a role.} && (3)\\
Permission &\sqsubseteq (\geq 0\ ConstraintAssignment^{-}.\top) \sqcap && (4)\\
&\quad (\leq 1 ConstraintAssignment^{-}.\top) &&
\end{aligned}$$

(ii) for authorization constraint a in m, $a \sqsubseteq AuthorizationConstraint$, where a and $AuthorizationConstraint$ are concepts representing the classes with the same names; (iii) for permissions p_1 and p_2, $p_1 \sqsubseteq Permission$ and $p_2 \sqsubseteq Permission$; (iv) for the associations between a and permissions p_i, $i \in \{1, 2\}$:

$$\begin{aligned}
&ConstraintAssignment\text{-}a\text{-}p_i \sqsubseteq ConstraintAssignment,\\
&\top \sqsubseteq \forall ConstraintAssignment\text{-}a\text{-}p_i.a \sqcap \forall ConstraintAssignment\text{-}a\text{-}{p_i}^{-}.p_i,\\
&\ a \sqsubseteq (\equiv 1\ ConstraintAssignment\text{-}a\text{-}p_i.p_i),\\
&p_i \sqsubseteq (\geq 0\ ConstraintAssignment\text{-}a\text{-}{p_i}^{-}.\top) \sqcap\\
&\qquad (\leq 1 ConstraintAssignment\text{-}a\text{-}{p_i}^{-}.\top).
\end{aligned}$$

The knowledge base described above is *inconsistent* because axioms 1 to 4 constrain role *ConstraintAssignment* to *exactly* one *Permission* for each *AuthorizationConstraint*. This may *not* be the case if there are individuals from both concepts p_1 and p_2 related to an individual of a.

For the second scenario, let us consider a model m containing an instance of *Action*. According to SecureUML+Component metamodel, there must exist a one-to-one association between a given *Action* and a *Resource*, which is *not* the case in our scenario. Due to the open world assumption, DL reasoning would *not* identify this violation. As we mentioned before, the open world assumption allows us to identify inconsistencies on *given* information. Nothing can be said if the information is not there. This is where OCL validation comes into place. The application of the invariant that constrains the cardinality on the association between *Action* and *Resource* would fail for m as the collection returned by navigating through the association between *Action* and *Resource* from a would produce an empty collection when it should be of size 1.

It should be straightforward to see that the verification and validation illustrated in this section applies to the model transformation context with transformation contracts described in Section 5.2 as the model one wants to verify and validate is the model resulting from the join of the source and target models. Therefore, the same techniques that one uses to verify and validate a model as an instance of a metamodel can be used to verify and validate a model transformation when specified by a transformation contract since it is a transformation model given by the joined model of the source and target models of a transformation.

6 A Design Pattern for the Implementation of Model Transformations with Transformation Contracts

We have defined a *design pattern* that captures the general process of model transformations with transformation contracts described in Section 3. The design pattern enforces the verification and validation at the different points that they must occur in a model transformation, that is, the analysis of: (i) the source model before the model transformation is applied, (ii) the target model after the transformation is applied and (iii) both source and target models and the associations between them also after the application of the model transformation. By analysis we mean both verification and validation, applying DL reasoning and OCL validation as described in Section 5. As a matter of fact, the design pattern, as well as the model transformation process it implements, is general enough to incorporate new analysis techniques and not only DL reasoning and OCL validation for different modeling languages when the proper encodings are defined, of course.

Figure 4 presents a class diagram of our proposed design pattern. In the pattern, a *Domain* represents a modeling language which interacts with a *ModelManager*, responsible for model persistency, and validators responsible for model analysis. Each *Domain* has a parser from the XMI standard representation, that is, the *Domain*'s concrete syntax, to its metamodel, which is the *Domain*'s abstract syntax. The joined metamodel of a transformation contract is represented by class *JoinedDomain* which references the two instances of *Domain* it relates, named source and target. The class *TransformationContract* declares a *static* method *transform* that executes the model transformation process that we have explained in Section 5. It is static because it is always the same behavior, independently of the actual domains that a particular model transformation relates. The instances of *Domain* will perform the "real" work since parsing, pretty printing and the encodings to the different formalisms are either implemented in methods within classes that inherit from *Domain* or that a *Domain* delegates to its instances.

Figure 5 depicts the application of the design pattern for model transformations with transformation contracts to the model transformation from SecureUML+ComponentUML to JAAS. (As mentioned before, the model transformation also uses aspect-oriented model elements, which are out of the scope of this paper, but this is the reason why the acronym AAC appears in the model.) SecureUML and SecureUML+ComponentUML are coded as different domain classes. The latter extends the former with metaclasses and associations making the action and resource hierarchies more concrete, as explained in Section 4. Moreover, each domain has its own validator class that implements the encoding to the proper reasoner. For DL reasoning, we use

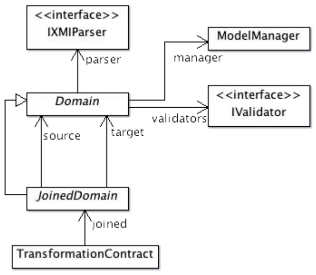


Fig. 4. A design pattern for model transformations with transformation contracts

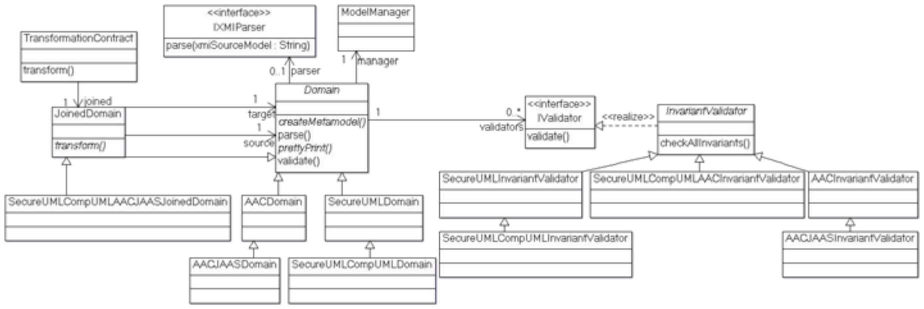


Fig. 5. Applying the design pattern to the SecureUML to JAAS model transformation

the Pellet[2] reasoner and for OCL execution we use EOS[3], which is also used to manage model persistence. Implementing a model transformation as an application of our proposed design pattern enforces the implementation of a rigorous model transformation as different verification and validation techniques can be applied.

7 Final Remarks

We are developing and applying a general technique for the rigorous specification, verification and implementation of model transformations using the concept of transformation contracts. A transformation contract is essentially a transformation metamodel that relates metamodels and a set of properties over the transformation metamodel. Implementations of model transformations are realized as an application of a design pattern that enforces our proposed model transformation process. In this paper we have used the standardized metalanguages of UML class diagrams, OCL for the specification of metamodels and invariants over them and Description Logic to verify consistency. However, our approach is not coupled with any particular choice of metalanguages and different

[2] http://clarkparsia.com/pellet/
[3] http://www.bm1software.com/eos/

metanotations and reasoners may be employed in the development of a model transformation. We plan to continue our work by integrating different automated analysis techniques to our model transformation process and to apply our approach to industrial case studies.

References

1. Akehurst, D.H., Kent, S.: A relational approach to defining transformations in a metamodel. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 243–258. Springer, Heidelberg (2002)
2. Baader, F., Diego Calvanese, D.M., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook. Cambridge University Press (2003)
3. Basin, D., Clavel, M., Doser, J., Egea, M.: Automated analysis of security-design models. Inf. Softw. Technol. 51(5), 815–831 (2009)
4. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From uml models to access control infrastructures. ACM Trans. Softw. Eng. Methodol. 15(1), 39–91 (2006)
5. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artif. Intellig. 168, 70–118 (2005)
6. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model Transformations? Transformation Models! In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006)
7. Braga, C.: From access control policies to an aspect-based infrastructure: A metamodel-based approach. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 243–256. Springer, Heidelberg (2009)
8. Braga, C.: A transformation contract to generate aspects from access control policies. J. of Software and Systems Modeling (2010), doi:10.1007/s10270-010-0156-x
9. Braga, C., Hæusler, E.H.: Lightweight analysis of access control models with description logic. Innov. in Systems and Soft. Eng. 6, 115–123 (2010)
10. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: Proc. of OCL and Model Driven Eng. Work., pp. 69–83 (2004)
11. Clavel, M., Egea, M., de Dios Miguel Angel, G.: Building an efficient component for OCL evaluation. ECEASST 15 (2008)
12. Comicio, T.: A transformation contract approach for model-driven security. Master's thesis, Universidade Federal Fluminense (2011)
13. Egea, M.: An Executable Formal Semantics for OCL with Applications to Model Analysis and Validation. PhD thesis, Universidad Complutense de Madrid (2008)
14. Gorp, P.V., Janssens, D.: Cavit: a consistency maintenance framework based on transformation contracts. In: Transformation Techniques in Soft. Eng., Dagstuhl Seminar Proc., vol. 05161 (2006)
15. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained. Addison-Wesley, Reading (2003)
16. OMG. MOF QVT final adopted specification, omg adopted specification ptc/05-11-01 (2005)
17. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)

Modular Embedding of the Object Constraint Language into a Programming Language

Fabian Büttner and Martin Gogolla

University of Bremen, Computer Science Department,
Database Systems Group
{green,gogolla}@tzi.de

Abstract. The Object Constraint Language (OCL) is a well-accepted ingredient in model-driven engineering and accompanying modeling languages like UML (Unified Modeling Language) or EMF (Eclipse Modeling Framework) which support object-oriented software development. Among various possibilities, OCL offers the formulation of state invariants and operation contracts in form of pre- and postconditions. With OCL, side effect free query operations can be implemented. However, for operations changing the system state an implementation cannot be given within OCL. In order to fill this gap, this paper proposes the language SOIL (Simple OCL-like Imperative Language). The expression sublanguage of SOIL is identical to OCL. SOIL adds well-known, traditional imperative constructs. Thus by employing OCL and SOIL, it is possible to describe any operation in a declarative way and in an operational way on the modeling level without going into the details of a conventional programming language. In contrast to other similar approaches, the embedding of OCL into SOIL is done in a new, careful way so that elementary properties in OCL are preserved (for example, commutativity of logical conjunction). The paper discusses the major criteria of a conservative embedding of OCL into SOIL. SOIL has a sound formal semantics and is implemented in the UML and OCL tool USE (UML-based Specification Environment).

1 Introduction

Modeling languages like UML (Unified Modeling Language) or EMF (Eclipse Modeling Framework) play a central role in object-oriented software development and rely on a model-centric approach for development in contrast to traditional code-centric approaches. One main idea when using models is to find and to formulate central structural and behavioral properties of the system under development in an abstract, implementation independent way. Visual modeling notations are typically enriched by the textual Object Constraint Language (OCL) [31,8] which combines elements of first order predicate logic with object navigation. OCL allows the developer to formulate properties of a model that cannot be expressed in the visual notation. Typical applications of OCL are the formulation of class invariants (to express structural properties) and pre- and postconditions for operations as well as guards for state charts (to express behavioral properties).

While there are several visual possibilities like state charts or activity diagrams for modeling behavior, there is no way which allows the developer to express imperative algorithms in textual form. However, there are two important areas that require such a complementary textual notation for models, because they involve a considerable number of imperative algorithms even on the modeling level: executable models and model transformations. Both areas typically combine a visual notation like state charts and graph transformations with imperative formulations of algorithms.

– The Executable UML approach [23,22] describes UML models that can be actually executed. This is achieved by providing Moore state charts for all operations of the model. Furthermore, a textual action language is used to describe the effects of the states in the state machines.
– A second application of textual imperative languages within modeling is found in the Model-Driven Architecture (MDA) [26]. The OMG Query, Views, Transformation (QVT) specification [28] describes several layers to transform abstract models into more specific models. At its core, it offers a textual imperative language, ImperativeOCL, which is based on OCL.
– Apart from the two former approaches, imperative descriptions within models can also be important as part of a general refinement process, to provide precise operational semantics to selected aspects of a model. Executable UML can be regarded as a special case of this.

Thus, one the one hand side, there is OCL, which has already proven to be a valuable expression language with broad support of tools. On the other hand, there are several areas that require an imperative language for and within models. Since imperative statements like assignments or conditionals require expressions, the idea to use OCL as an expression language within an imperative programming language naturally comes up.

Indeed, there is a number of imperative languages that reuse OCL as an expression language. However, if one looks at the reuse of OCL in these approaches in depth, there are different understandings of reuse. ImperativeOCL is an example for a kind of weak OCL reuse that prohibits the direct reuse of OCL tools and it is also an example for a language that introduces semantic problems for OCL expressions [5]. As an alternative, we developed the language SOIL (Simple OCL-based Imperative Language) [3]. SOIL has a sound formal semantics and is type safe. SOIL is implemented in the UML-based Specification Environment (USE) [11].

It has to be stressed here that none of the approaches listed above proposes an imperative language for UML as an appropriate general purpose programming language, and nor do we propose this for UML plus SOIL. Instead, the imperative language complements the other modeling paradigms for specific purposes.

In the present paper, we will use SOIL as a showcase to illustrate the major criteria for reusing OCL in an imperative programming language. The rest of the paper is structured as follows. We first motivate why OCL could be reused in an imperative programming language in Sect. 2. In Sect. 3, we give an example of how SOIL is applied to specify the semantics for operations in models. In Sect. 4 we discuss general criteria for the reuse of OCL in an imperative programming language. When necessary, we refer to SOIL to illustrate a modular kind of reuse. Section 5 shortly highlights the consequences of a modular embedding in terms of language expressiveness. We conclude our paper in Sect. 6.

2 Motivation for Reusing OCL

In the context of model-driven engineering and model-transformation, there are several reasons to reuse OCL. Formal approaches such as Executable UML and MOF QVT require precise operational descriptions that cannot always be expressed reasonably using only visual notation. Textual imperative languages are required to fill this gap. The official OMG language ImperativeOCL extends OCL by so-called imperative expressions to suit this need. Other approaches combining OCL or an OCL-like language with imperative programming include ATL [15], EOL [17] and OCL4X [14].

There are several reasons to build these languages on top of OCL. First of all, we can assume OCL to be already known in context where these languages are used. Developers familiar with modeling languages typically know OCL already. Thus, learning the respective imperative language becomes easier when the expression language is already known. This is true in particular as these languages are typically rather lightweight. They do not aim to compete against general purpose languages such as Java or C#.

Another reason for reusing OCL is the possibility to reuse existing OCL tools: The implementation of a programming language based on OCL can be simplified if one can avoid to deal with expressions again. The infrastructure for UML and EMF models and OCL expressions is already available in several tools. The long list of publicly available OCL tools includes the Dresden OCL toolkit [13], the OCL Environment (OCLE) [7], the Eclipse Model Development Tools (MDT) Eclipse MDT OCL [21], KMF [1], the Octopus tool [16], RoclET [30], and the UML-based Specification Environment (USE) [11].

Furthermore, the scientific community has developed a number of formal approaches that deal with OCL expressions and OCL-annotated models. These approaches include expression transformation (e. g., in [20,6,4]), expression analysis (e. g., in [9]), reasoning (e. g., in [2]), and model checking (e. g., in [10,19]). These results can be employed further, if OCL is used as an expression language within an imperative programming language.

For these reasons, we think that there are strong arguments for reusing OCL in imperative programming. However, as we will point out in Sect. 4, we have to be careful in the definition of an OCL-based imperative language. OCL has to be embedded in a modular way when one wants to take advantage of the mentioned profits.

3 SOIL by Example

In this section we give a concrete example for using an imperative programming language for UML models: The language SOIL as part of the UML-based Specification Environment (USE). USE supports the modeler in two ways: (1) prototypical model states for OCL-annotated UML models can be validated against structural constraints (including OCL invariants), and (2) prototypical model executions can be validated against dynamic constraints (i. e., OCL pre- and postconditions). Previously, there was no universal means to specify imperative programs for OCL-annotated models employing general loops, operation calls and recursion. This gap was filled by SOIL. Using SOIL, imperative definitions can be given for the operations of a model, and the imperative definitions can be validated against the structural and dynamic constraints of the model.

The extended USE tool now enables stepwise refinement from a declarative model (pre- and postconditions) towards an operational model (operation implementation) in an integrated model-based environment. The following short example shows how SOIL is used to perform this refinement step. Consider the class diagram in Fig. 1. In this project world, companies employ workers and carry out projects. Workers bring certain qualifications (e.g., programming) and projects require certain qualifications. In order for a project to become active, it must have members for all required qualifications. In this class diagram, we have only one non-query operation, schedule(), to assign workers to projects. A good implementation of schedule() will ensure a good use of the company's human resources (ideally, carry out as many projects as possible).

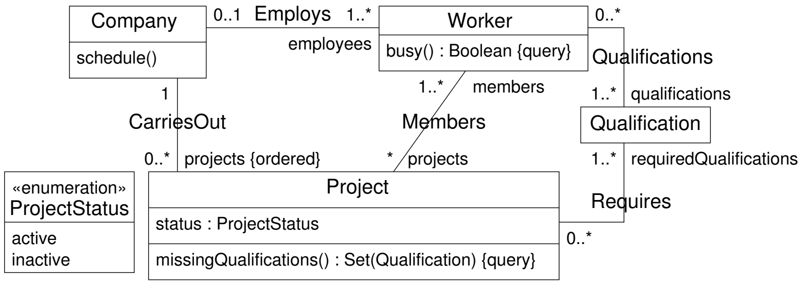


Fig. 1. Project World

Some properties of this operation are further specified in a declarative way by OCL postconditions as shown in Listing 1.1: After scheduling projects, it has to be ensured that no active project lacks any qualifications and no employee is working in two active projects at the same time. The listing also shows the definition of the two query operations missingQualifications() and busy(). These side effect free operations are defined straightforward by OCL expressions.

Obviously, several implementations of schedule() will full the above pre- and postconditions. The USE tool allows us define schedule() using SOIL statements. Giving an initial state, all SOIL defined operations can be invoked. Recursive invocation is supported, as well. During the animation of the model, all structural and dynamic constraints are checked. In our example, the execution of the schedule() operation is validated against the above postconditions. We can compare this functionality to programming languages that support design-by-contract (such as Eiffel [24]). However, in our case we are still in the context of the UML object model. In particular, OCL expressions can be used within our imperative definition.

```
context Project def: missingQualifications() :
    Set(Qualification) =
  self.requiredQualifications -
      self.members.qualifications->asSet

context Worker def: busy() : Boolean =
  self.projects->exists(p | p.status = #active)

context Company::schedule()
  post activeProjectsHaveRequiredQualifications:
    self.projects->forAll(p | p.status = #active implies
      p.missingQualifications()->isEmpty)
  post employeesNotOverloaded:
    self.employees->forAll( w | w.projects->select(p |
      p.status = #active)->size <= 1)
```

Listing 1.1. Declarative specification of Company::schedule

Listing 1.2 shows a very simple imperative version of schedule(). We can see that the SOIL provides typical flow control constructs (*for*-loop, *if*-statement). Within these statements, OCL expressions are used to describe the parameters (e.g., the range for the iteration and the condition for the *if*-statement. Statements to manipulate the system state are available (in the above example: link insertion and attribute assignment). The semantics of these statements is straightforward.

```
context Company def: schedule() =
  for w in self.employees do
    for p in self.projects do
      if p.missingQualifications()
           ->intersection(w.qualifications)->notEmpty then
        insert (p, w) into Members;
        if p.missingQualifications()->isEmpty and not
            w.busy() then
          p.status := #active
        end
      end
    end
  end
```

Listing 1.2. Operational specification of Company::schedule

USE processes all aspects of the project world model, as defined in this paper: The static structures can be instantiated (i.e., objects and links can be created). This can happen either manually, using the USE *Generator* (see below), or using SOIL statements. Then, the instantiated system state can be validated against all structural constraints of the model. Regarding the dynamic aspects of a model, any manually provided flow of actions (i.e., a particular sequence of state changes) as well as any execution of a SOIL-defined operation can be validated against the dynamic constraints of the model (i.e., against the pre- and postconditions).

The tool can be employed to validate that our very simple implementation of schedule() conforms to the postconditions as follows. For given initial system state, USE can check if a particular execution of schedule() conforms to its two postconditions. A sufficient coverage of test cases can be provided by means of a surrounding SOIL program, or by employing the *Generator* language [11] of USE (the *Generator* implements a backtracking search to yield valid instances of the model). This kind of validation can be seen as systematic testing, in contrast to a formal verification of correctness.

While the above schedule() conforms the postconditions, it is not an optimal implementation, since it will not result in a maximum number of projects being active. We can construct a more sophisticated version that schedules and reschedules project members to achieve an optimal number of active projects. However, such an implementation is much more complex and, therefore, error prone. The integrated descriptive and operational specification in USE with OCL and SOIL allows a smooth and step-wise refinement process from the descriptive model to an actual imperative model, guiding the developer by validating the models through animation against the constraints.

4 Embedding of OCL into SOIL

In the previous section, we gave an example for an imperative programming language for UML models. As several other related languages such as ImperativeOCL, it reuses OCL for expressions. However, not all of these languages realize the benefits we gave as reasons for reusing OCL in Sect. 2.

In this section, we now discuss several concepts of imperative programming languages from the perspective of reusing OCL. We start with a short recapitulation of the formalization of OCL expressions, then we inspect the amalgamation of statements, local variables, operation invocation, and state manipulation. As we will see, the major pitfalls in a successful modular reuse of OCL lurk in the amalgamation of statements and expressions and in an indeterminate treatment of operations with side effects and query operations.

For each of the mentioned concept we provide the corresponding piece of SOIL to illustrate a safe and modular reuse. A complete guide to SOIL, including a formal definition of the language as well as proofs for the type soundness can be found in [3].

4.1 Formal Representation of OCL Expressions

We shortly sketch the formal definitions for UML static structure models and OCL expressions. These definitions have been originally provided in [29] and are now contained in the OCL specification [27]. For the objective of this paper, a complete depiction of the formalization is not necessary as we only need the general concepts in the following.

The object model

$$\mathcal{M} = (\textsc{Class}, \textsc{Att}_c, \textsc{Op}_c, \textsc{Assoc}, \mathrm{associates}, \mathrm{roles}, \mathrm{multiplicities}, \prec)$$

is the formal representation of the major concepts UML provides for static structure modeling (say class diagrams). It contains all classes along with their attributes, operation signatures, associations, and generalization relationships. The set μ denotes the set of all instances of $\mathcal{M}$. Thus, a system state $\sigma \in \mu$ describes a set of objects, links, and attribute values.

For the formalization of OCL expressions, we require a data signature over $\mathcal{M}$ which is a structure

$$\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$$

where $T_{\mathcal{M}}$ is the set of all types over $\mathcal{M}$. This includes primitive types, user types (in particular, classes), and all collection types can be constructed by the OCL collection type constructors. The relation $\leq$ is the type hierarchy over $T_{\mathcal{M}}$. The set $\Omega_{\mathcal{M}}$ contains the set of all query operations (operations without side effects), and thus corresponds to a subset of OP_c.

The semantics of $\Sigma_{\mathcal{M}}$ is as follows. $I(T_{\mathcal{M}})$ assigns each type $t \in T_{\mathcal{M}}$ an interpretation $I(t)$ (the domain of t). $I(\leq)$ implies for all types $t', t \in T_{\mathcal{M}}$ that $I(t') \subseteq I(t)$ if $t' \leq t$. $I(\Omega_{\mathcal{M}})$ assigns each operation $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_{\mathcal{M}}$ a total function $I(\omega) : \sigma \times I(t_1) \times \cdots \times I(t_n) \to I(t)$.

Given the data signature $\Sigma_{\mathcal{M}}$, we can formalize the set Expr of all OCL expressions that exists over $\Sigma_{\mathcal{M}}$. For each expression $e \in$ Expr, the function $\text{free} :$ Expr $\to$ Var determines the free variables of e (Var being the set of all typed variables).

The interpretation of an expression $e \in$ Expr is given by a function $I[\![\, e \,]\!]$ which assigns a value to each pair $\tau = (\sigma, \beta)$ of a system state σ of $\mathcal{M}$ and a variable assignment $\beta :$ Var $\to I(t)$.

4.2 Statements

Imperative programming languages typically refer to their smallest standalone elements as statements. The effect of such a statement is determined by its effect on the process environment (the state). For imperative languages that work on object models, the state at least contains the available objects, links, and attribute values, as well as a representation of the variable assignments.

If we want to describe the semantics of an imperative language in a similar fashion as for OCL, then we have to describe the interpretation for each statement s of that language by an interpretation function. A minimalistic interpretation function for statements is a function $I[\![\, s \,]\!]$ which assigns each pair (σ, β) of a system state σ and a variable assignment β a new pair (σ', β'). If we furthermore want a statement to have a value in a functional sense, we require an interpretation that assigns each pair (σ, β) a triple (σ', β', y) where y is the functional value of statement s.

Statements having a functional value may also occur where an expression is expected. Several statements in common programming languages have this kind of semantics, for example the assignment statement $b = a$ in Java which (1) leads to a new environment (with b having a new value) and (2) has a functional value (the value of a). It can be used as an expression as well, therefore a statement like $c = (b = a)$ is valid in several programming languages.

However, for a modular reuse of OCL, it is important to keep statements and OCL expressions clearly separated. We will use the language ImperativeOCL to illustrate the problems that result from an amalgamation of statements and OCL expressions.

ImperativeOCL defines several new kinds of OCL expressions. These new expressions are called imperative expressions and have a combined functional resp. imperative semantics as explained above. In the ImperativeOCL metamodel, the imperative expressions are introduced as subclasses of *OclExpression* (and therefore, imperative expressions extend the set of OCL expressions).

In particular, the *compute* expression can be used to capture the result of a sequence of imperative statements as a functional value. In ImperativeOCL, the following expression has the value 6 $(1 + 2 + 3)$:

```
1 + compute(b : Integer) { a := 1; b := a + 1 } + 3
```

The compute expression declares a local variable and contains a sequence of imperative expressions. The value 2 of the above compute expression is determined by the final value of b after executing the statements of the body. If we assume the second variable a to be declared somewhere before, the compute expression also has an effect that is visible outside the compute expression, as a (possibly) new value (1) will be assigned to a after the evaluation of the compute expression.

Now we use a more complex example. Assume *true* has been assigned to the variables a and b before, and notice that the imperative assignment expression $x := y$ of ImperativeOCL has the same value semantics as discussed above:

```
compute(c:Boolean) {
  if ((a:=false) and (b:=false)) { ... }; c := a }
```

The value of this compute expression is false (it returns the value of c at the end of the block). The interpretation, however, becomes less obvious if we change the last assignment:

```
compute(c:Boolean) {
  if ((a:=false) and (b:=false)) { ... }; c := b }
```

The interpretation of this compute expression depends on how we define the imperative semantics of the logical connectives. Given Boolean expressions e_1 and e_2, we have at least two choices to define $I[\![\, e_1 \text{ and } e_2 \,]\!](\sigma, \beta)$:

1. Lazy evaluation semantics like in Java or C (returns true for the above example):

$$I[\![\, e_1 \text{ and } e_2 \,]\!](\sigma, \beta) = \begin{cases} I[\![\, e_2 \,]\!](\sigma', \beta') & \text{if } y = \text{true} \\ (\sigma', \beta', y) & \text{otherwise} \end{cases}$$

   where $(\sigma', \beta', y) = I[\![\, e_1 \,]\!](\sigma, \beta)$. Under this semantics (also called short-circuit evaluation) the right-hand side of the *and* operator is not evaluated unless the left-hand side evaluates to *true*. Therefore, b stays *true*.

2. Strict evaluation semantics (returns false for the above example):

$$I[\![\, e_1 \text{ and } e_2 \,]\!](\sigma, \beta) = \begin{cases} (\sigma'', \beta'', \text{true}) & \text{if } y_1 = \text{true} \wedge y_2 = \text{true} \\ (\sigma'', \beta'', \text{false}) & \text{otherwise} \end{cases}$$

   where $(\sigma', \beta', y_1) = I[\![\, e_1 \,]\!](\sigma, \beta)$ and $(\sigma'', \beta'', y_2) = I[\![\, e_2 \,]\!](\sigma', \beta')$. Under this semantics, both sides of the *and* operator are always evaluated. Therefore, *false* is assigned to b.

There is no rule on short-circuit evaluation in OCL. OCL, which can be regarded as a kind of first order predicate logic, does not need such a rule. An optimizing OCL compiler might even decide to short-circuit evaluate the second operand first if this seems reasonable.

However, in order to have a clear semantics, ImperativeOCL implicitly requires a decision on this question. Similar issues regard the commutativity of operators etc. Of course these decisions can be made for ImperativeOCL, but they may be inappropriate for other applications of OCL. And, existing OCL tools may have differing implementations and may be therefore unusable to implement ImperativeOCL.

A more general argument against the amalgamation of statements and expressions is that OCL expressions are no longer side effect free by introducing *ImperativeExpression* as a subclass of *OclExpression*. In our understanding, this breaks a fundamental property of the *OclExpression* class. Therefore the ImperativeOCL metamodel breaks the subtype substitution principle. The direct result is that formal approaches such as expression transformations, expression analysis, reasoning, and model checking cannot longer be applied to OCL expressions in the context of the ImperativeOCL extension.

Therefore, we require a strict distinction of statements and OCL expressions for a modular reuse of OCL. Fig. 2 depicts this requirement on the level of the language metamodels. Notice that, from the perspective of modular reuse, an imperative programming language might add further kinds of expressions which are not OCL. However, these expressions must not occur as OCL expressions.

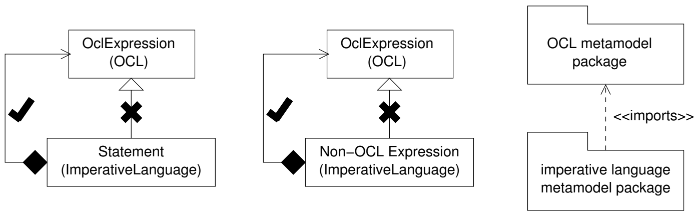

OclExpression (OCL)
Statement (ImperativeLanguage)
OclExpression (OCL)
Non−OCL Expression (ImperativeLanguage)
OCL metamodel package
<<imports>>
imperative language metamodel package

Fig. 2. Modular Embedding

A similar argumentation for composition and modularity of domain specific languages can be found in [18] and [12]. It is also aligned with [25] in the sense that side effected non-modular extensions of OCL should be avoided.

SOIL Example. For the reasons given, statements are clearly separated from OCL expressions in SOIL. To illustrate the formalization of statements in SOIL, we show how the syntax and semantics of the imperative *if-then-else* are defined.

The syntax is defined as follows (typing rules omitted, as explained below): If $e \in \mathrm{Expr}_{\mathrm{Boolean}}$ and $s_1, s_2 \in \mathrm{Stmt}$ then

$$\text{if } e \text{ then } s_1 \text{ else } s_2 \text{ end} \in \mathrm{Stmt}.$$

We can see that this kind of statement *contains* an expression. But despite syntactic similarities to the functional *if-then-else* of OCL, the imperative *if-then-else* is a completely different entity: The definitions of Expr and $I[\![e]\!]$ are not changed or extended by SOIL. The imperative language defines a new set of statements Stmt, which is disjoint with Expr.

The meaning of each statement $s \in \text{Stmt}$ is given by an interpretation function $I[\![\, s \,]\!]$ which assigns each pair (σ, ζ) of a system state and a variable assignment a new pair (σ', ζ'). Notice that, for technical reasons, we distinguish the imperative variable assignments ζ (which actually are a stack structure) and the variable assignments β (used to evaluate OCL expressions).

The semantics of the *if* statement is defined as follows.

$$I[\![\, \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ end} \,]\!](\sigma, \zeta) := \begin{cases} I[\![\, s_1 \,]\!](\sigma, \zeta) & \text{if } I[\![\, e \,]\!]\big(\sigma, \text{binding}(\zeta)\big) = \text{true} \\ I[\![\, s_2 \,]\!](\sigma, \zeta) & \text{otherwise} \end{cases}$$

Corresponding to the syntactic containment of OCL expressions as part of statements, the interpretation function for OCL expressions occurs within the above definition of the interpretation function for the *if* statement. The condition expression e is evaluated in the same context (state, variables) as the statement. To pass the variable assignments from $I[\![\, s \,]\!]$ to $I[\![\, e \,]\!]$ we require a transformation binding to map between the different notions of variable assignments in SOIL and OCL (see next subsection).

All kinds of statements in SOIL are defined in this manner.

On Typing Rules. We omitted the typing rules of SOIL in this paper for the reasons of space and simplicity, as they are not relevant for the discussion of modularity. We shortly sketch their structure in this subsection. To define the set of statements Stmt, the complete formal definition of the syntax of SOIL assigns three sets to each statement s in Stmt: $\text{free}(s)$ is the set of typed variables which must be present in a variable binding in order to execute the statement s, $\text{assigned}(s)$ is the set of all variables that *might* be assigned by s, and $\text{bound}(s)$ is the set of all variables that *definitely* have a value of a certain type after executing s. Using these three sets, we proved in [3] by induction over the structure of statements that the interpretation $I[\![\, s \,]\!](\sigma, \zeta)$ is total and well-defined given that ζ is a variable stack over σ and ζ containing properly typed values for all free variables of s. As for the syntax in general, the typing rules of SOIL do not affect the typing rules of OCL. The type system of SOIL, however, guarantees that no invalid expressions (such as `'a' + 1`) can occur as part of a statement.

4.3 Local Variables

Variable assignment is a core concept available in all imperative programming language. When statements contain OCL expressions, the assignments of previous statements will be visible for the evaluation of OCL expressions in subsequent statements. Consider the imperative program:

```
a := 1; b := a + 1.
```

The OCL expression in the second statement has one free variable a. A value for a will be available after the execution of the first statement (as a furthermore has the right type, the above concatenation of statements is even type sound). This relationship is depicted in Fig. 3. In SOIL the mapping from the imperative variable environment (which is a stack) to the variable assignment (which is a flat mapping) required for the evaluation of OCL expressions is realized by the *binding* operation which already occurred above.

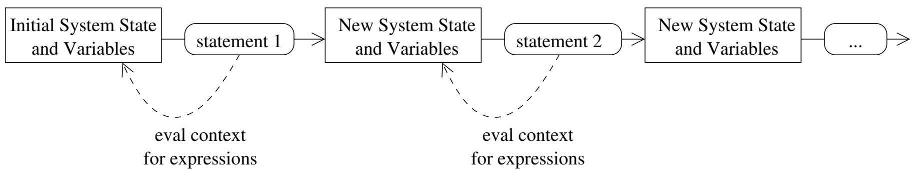

Initial System State and Variables
statement 1
New System State and Variables
statement 2
New System State and Variables
...
eval context for expressions
eval context for expressions

Fig. 3. Evaluation Chain for Statements and Expressions

Technically, *binding* makes the assignments in the top-most stack frame in ζ available as a flat mapping from typed variables to values.

There are no particular obstacles regarding local variables w.r.t. modularity of OCL. However, if we want static type checking, it is important to ensure that correctly typed values are available for all free variables in the OCL expressions that are part of our statements (we provide such a type system in SOIL).

4.4 Operations with Side Effects

The application of operations with side effects within OCL expressions constitutes a similar problem as the amalgamation of statements and OCL expressions. While the interpretation of a query operation is a value (see Sect. 4.1), the interpretation of an operation with side effects yields a new state (and possibly a value). For a modular reuse of OCL we cannot allow the second one to occur in OCL expressions.

In order not to stretch short-circuit evaluation or commutativity of relational operations for the explanation, again, we take a look on the *let* expression in OCL. This expression substitutes an expression for a variable. As for predicate logic, the following important equivalence rule holds for OCL:

$$I[\![\, \text{let}\ v : T = e_1\ \text{in}\ e_2 \,]\!](\sigma, \beta) = I[\![\, e_2\{v/e_1\} \,]\!]\big(\sigma, \beta\big).$$

However, this rule is broken if we allow operations with side effects within OCL expressions. Assume a class Person with attributes firstName and lastName. Consider an operation newPerson():

```
def: newPerson(firstName : String, lastName : String):Person =
  w := new Worker;
  w.firstName := firstName;
  w.lastName := lastName;
  return w
```

Obviously, the interpretation of

```
let w : Worker = newWorker('Bob', 'Builder') in
  w.lastName.concat(', ').concat(w.firstName)
```

is different from the interpretation of

```
newWorker('Bob', 'Builder').lastName.concat(', ').concat(
  newWorker('Bob', 'Builder').firstName)
```

which will create two Worker objects.

As mentioned above, these problems can be constructed in several ways if we allow operations with side effects in OCL expressions. Therefore, we require a distinction between query operations and operations with side effects. Within OCL expressions, only query operations must be used. Otherwise, we run into the same problems mentioned in Sect. 4.2. Consequently, an imperative language must include a dedicated means to invoke operations with side effects.

SOIL Example. The language provides specific statements to invoke operations with side effects. Here, we show the form which invokes an operation that has a return value (another statement is available to invoke operations without return value). Its syntax is as follows: If $e_1 \in \mathrm{Expr}_{t_1}, \dots, e_n \in \mathrm{Expr}_{t_n}$, $v \in$ Varname, and $\overline{\omega} : (v_1 : t_1, \dots, v_n : t_n \to t) \in \overline{\Omega}_{\mathcal{M}}$ then

$$v := \overline{\omega}(e_1, \dots, e_n) \in \text{Stmt}.$$

The most important point here is that the operation to be invoked has to be in the set of the operations with side effects $\overline{\Omega}_{\mathcal{M}}$, whereas query operations (which can be occur in OCL expressions) are in $\Omega_{\mathcal{M}}$ (c.f. Sect. 4.1).

Given Z being the set of all variable assignments ζ, the semantics of each $\overline{\omega} : t_1 \times \dots \times t_n$ in $\overline{\Omega}_{\mathcal{M}}$ is a total function

$$I(\overline{\omega}) : \mu \times Z \times I(t_1) \times \dots \times I(t_n) \to \mu \times Z$$

that assigns to the current system state, variable assignments, and parameters a new system state and a new variable assignment.

The semantics of s is then given as follows. Let $x_1, \dots, x_n = I[\![\, e_1 \,]\!]\big(\sigma, \mathrm{binding}(\zeta)\big), \dots, I[\![\, e_n \,]\!]\big(\sigma, \mathrm{binding}(\zeta)\big)$, then

$$I[\![\, v := \overline{\omega}(e_1, \dots, e_n) \,]\!](\sigma, \zeta) := (\sigma', \zeta'\{v/z\})$$

where $(\sigma', \zeta', z) = I(\overline{\omega})(\sigma, \zeta, x_1, \dots, x_n)$.

4.5 State Manipulation Statements

Imperative languages that operates on UML models typically at least provide the following capabilities: object creation and destruction (unless a garbage collection approach is applied), link manipulation, and attribute assignment. Most of these kinds of statements have parameters (e.g., to determine the elements of a link) which can be given by OCL expressions. If the modularity aspects highlighted so far are obeyed, there are no further obstacles w.r.t to a modular reuse of OCL. As for local variables, state manipulations in a previous statement have to be visible in OCL expressions as part of a subsequent statement.

5 Consequences of a Modular Embedding

In the previous sections we discussed several pitfalls for a modular embedding of OCL. If we avoid these pitfalls, we can achieve benefits stated in Sect. 2. Apart from the syntactical differences, languages that reuse OCL in a modular way (such as SOIL) can express programs in a similar way as languages that do not reuse OCL that way, like ImperativeOCL. Comparing SOIL and ImperativeOCL, both languages provide the full power of OCL for expressions.

There are, however, kinds of statements that cannot be translated one-to-one from ImperativeOCL to SOIL or to any language that obeys the rules given in the previous section. Specifically, these statements are statements that contain expressions that contain statements. Constructions such as

```
mySeq := Sequence{1,2,3}->collect( x |
   compute(y:Integer) {
     y := 0; Sequence{1..x}->forEach(z){ y := y + z }})
```

are not possible in SOIL and have to be decomposed into several steps in SOIL:

```
mySeq := Sequence(Integer){};
for x in Sequence{1,2,3} do
  y := 0; for z in Sequence{1..x} do y := y + z end;
  mySeq := mySeq->append(y)
end
```

Such amalgamation of expressions and statements have to be resolved in several steps in a modular embedding of OCL. Notice that this includes invocations of non-query (i.e., side effected) operations from OCL expressions: Assuming f and g to be operations with side effects that furthermore yield integer values, the following ImperativeOCL expression

```
result := f() + g() + 1
```

has to be rewritten in SOIL to

```
fVal := f(); gVal := g(); result := fVal + gVal + 1.
```

Of course a imperative language might allow the upper syntax as a shortcut for the lower syntax, but it is important to see that this effectively introduces a new set of non-OCL expressions as part of that imperative language (as depicted in the middle part of Fig. 2 on the metamodel level). While the syntax might look the same as OCL, existing OCL compilers (or interpreters) cannot be used to implement it, nor can we reuse other formal approaches for OCL expressions, for the reasons given in Sect. 3. Of course, one might believe this redundant approach to be viable for simple arithmetic expressions as above. However, we cannot see where to draw the line here: If we want to allow operations with side effects anywhere in a right-hand side expression of an assignment statement (for example), we have to re-implement the whole OCL syntax for this custom expression language (i. e., we don't anymore reuse OCL in the sense of Sect. 2). If we only allow certain (say, simple) expressions such as arithmetic expressions, it will probably appear inconsistent and confusing to the modeler, as operations with side-effects are allowed in some expressions only.

For these reasons, we believe that such a redundant approach should be avoided completely. The resulting general restrictions are the price we have to pay for a language that reuses OCL in a modular and comprehensive way. In the scope of programming (with) models, we think that the benefits by far outweigh this price. This holds in particular if we consider that we already have the full power of OCL expressions as part of the imperative language and therefore a lot of programming can be done in a functional manner.

6 Conclusion

In this paper we presented our understanding of a modular reuse of OCL in an imperative language. If languages embed OCL this way, the reuse of existing tools and libraries, of knowledge that developer already gained for OCL, and of formal methods for OCL expressions, is possible. Several OCL-based, or OCL-inspired languages fail to fulfill this requirements, in particular ImperativeOCL.

Based on this observation we developed the language SOIL. SOIL is a simple and unspectacular but complete imperative language that can be used to operationally specify UML models (i. e., to program (with) UML models). We used SOIL to illustrate the major drawbacks in the design of OCL-based imperative languages.

The intrinsic drawbacks of SOIL (and any other language that is based on OCL in a modular way) w. r. t. to monolithic languages such as ImperativeOCL regard amalgamation of expressions and statements. These constructs have to be decomposed in SOIL, which, in general, leads to larger programs. We believe, however, that for most of the (rather restricted) scenarios of programming with models, the benefits of reusing the well-known and established language OCL outweigh these extra efforts.

A number of topics will be addressed in future work. SOIL has already been employed in smaller case studies, but larger case studies must give feedback on the usability of the language. Further imperative constructs like more convenient loops and error handling should be addressed. SOIL is compliant with the UML Actions metamodel. Therefore, it could be used, in the Executable UML approach, in conjunction with state machines in order to create fully executable descriptions of a system.

References

1. Akehurst, D., Patrascoiu, O.: KMF (Kent Modeling Framework) OCL Library. website (2011), http://www.cs.kent.ac.uk/projects/ocl/tools.html (last visited February 10, 2011)
2. Brucker, A.D., Wolff, B.: HOL-OCL: A Formal Proof Environment for UML/OCL. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 97–100. Springer, Heidelberg (2008)
3. Büttner, F.: Reusing OCL in the Definition of Imperative Languages. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin (2011)
4. Büttner, F.: Transformation-Based Structure Model Evolution. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 339–340. Springer, Heidelberg (2006)
5. Büttner, F., Kuhlmann, M.: Shortcomings of the Embedding of OCL into QVT ImperativeOCL. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 263–272. Springer, Heidelberg (2009)

6. Cabot, J., Teniente, E.: Transformation techniques for OCL constraints. Science of Computer Programming 68(3), 179–195 (2007)
7. Chiorean, D., Pasca, M., Cârcu, A., Botiza, C., Moldovan, S.: Ensuring UML Models Consistency Using the OCL Environment. Electronic Notes in Theorethical Computer Science 102, 99–110 (2004)
8. Clark, A., Warmer, J. (eds.): Object Modeling with the OCL: The Rationale behind the Object Constraint Language. LNCS, vol. 2263, pp. 4–20. Springer, Heidelberg (2002)
9. Cuadrado, J.S., Jouault, F., Molina, J.G., Bézivin, J.: Deriving OCL Optimization Patterns from Benchmarks. ECEASST 15 (2008)
10. Distefano, D.S., Katoen, J.P., Rensink, A.: Towards model checking OCL. In: ECOOP 2000: Defining Precise Semantics for UML, Sophia Antipolis, France (June 2000)
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
12. Hudak, P.: Modular Domain Specific Languages and Tools. In: Proceedings of the Fifth International Conference on Software Reuse, pp. 134–142. IEEE Computer Society Press, Los Alamitos (1998)
13. Hußmann, H., Demuth, B., Finger, F.: Modular architecture for a toolset supporting OCL. Science of Computer Programming 44(1), 51–69 (2002)
14. Jiang, K., Zhang, L., Miyake, S.: Using OCL in Executable UML. ECEASST 9 (2008)
15. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
16. Klasse Objecten. The Klasse Objecten OCL Checker Octopus. website, www.klasse.nl/english/research/octopus-intro.html, Klasse Objecten (2005)
17. Kolovos, D.S., Paige, R.F., Polack, F.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
18. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In: Paige, R.F., Meyer, B. (eds.) TOOLS (46). LNBIP, vol. 11, pp. 297–315. Springer, Heidelberg (2008)
19. Krieger, M.P., Knapp, A.: Executing Underspecified OCL Operation Contracts with a SAT Solver. ECEASST 15 (2008)
20. Markovic, S., Baar, T.: Refactoring OCL annotated UML class diagrams. Software and System Modeling 7(1), 25–47 (2008)
21. Eclipse model development tools (MDT) project page. Website, http://www.eclipse.org/modeling/mdt/ (last visited February 10, 2011)
22. Mellor, S.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley (2002)
23. Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, Boston (2004)
24. Meyer, B.: Eiffel: The Language. Prentice-Hall, Englewood Cliffs (1992)
25. Siikarla, J.P.M., Selonen, P.: Combining OCL and Programming Languages for UML Model Processing. In: Schmitt, P.H. (ed.) Proceedings of the Workshop, OCL 2.0 – Industry Standard or Scientific Playground, vol. 102. Elsevier (2004)
26. OMG. MDA Guide Version 1.0.1. Object Management Group, Inc., Framingham, Mass, Internet (June 2003), http://www.omg.org
27. OMG. Object Constraint Language Specification, version 2.0 (Document formal/2006-05-01) (June 2006)

28. OMG. Meta Object Facility (MOF) 2.0 Query/Views/Transformation Specification (Document formal/08-04-03). Object Management Group, Inc., Framingham, Mass, Internet (2008), http://www.omg.org
29. Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, BISS Monographs, No. 14 (2002)
30. RoclET-Team. Welcome to RoclET. Website, http://www.roclet.org/ (last visited February 10, 2011)
31. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Object Technology Series. Addison-Wesley, Reading (2003)

Algebra of Monotonic Boolean Transformers

Viorel Preoteasa

Åbo Akademi University,
Department of Information Technologies,
Joukahaisenkatu 3-5 A, 20520 Turku, Finland

Abstract. Algebras of imperative programming languages have been successful in reasoning about programs. In general an algebra of programs is an algebraic structure with programs as elements and with program compositions (sequential composition, choice, skip) as algebra operations. Various versions of these algebras were introduced to model partial correctness, total correctness, refinement, demonic choice, and other aspects. We introduce here an algebra which can be used to model total correctness, refinement, demonic and angelic choice. The basic model of our algebra are monotonic Boolean transformers (monotonic functions from a Boolean algebra to itself).

1 Introduction

Abstract algebra is a useful tool in mathematics. Rather than working with specific models like natural numbers and algebra of truth values, one could reason in a more abstract setting and obtain results which are more general and applicable in different models. Algebras of logics are very important tools in studying various aspects of logical systems. Algebras of programming theories have also a significant contribution to the simplification of reasoning about programs. Programs are elements of an algebra and program compositions and program constants (sequential composition, choice, iteration, skip, fail) are the operations of the algebra. These operations satisfy a number of relations which are used for reasoning about programs. Kleene algebra with tests (KAT) [12] is an extension of Kleene algebra and it is suitable for reasoning about programs in a partial correctness framework. Various versions of Kleene algebras have been introduced, ranging from Kleene algebra with domain [8] and concurrent Kleene algebra [10] to an algebra for separation logic [7].

Refinement Calculus [1,2,6,15] is a calculus based on (monotonic) predicate transformers suitable for program development in a total correctness framework. Within this calculus various aspects of imperative programming languages can be formalized. These include total correctness, partial correctness, demonic choice, and angelic choice. Demonic refinement algebra (DRA) was introduced in [21,22] as a variation of KAT to allow also reasoning about total correctness. The intended model of DRA is the set of conjunctive predicate transformers and this algebra cannot represent angelic choice. General refinement algebra (GRA) was also introduced in [22], but few results were proved and they were mostly related to iteration. Although the intended model for GRA is the set of monotonic predicate transformers, GRA does not include the angelic choice operator. GRA has been further extended in [20] with enabledness and termination operators, and it was extended for probabilistic programs in [14].

The contribution of this paper is a different extension of GRA with a dual operator [9,4,5,6]. The intended model for our algebra is the set of monotonic Boolean transformers (monotonic functions from a Boolean algebra to itself). In GRA assertions (assumptions) are introduced as disjunctive (conjunctive) elements which have complement. Using the dual operator we characterize these assertion (assumptions) using a conjunction of (in)equations which is simpler than the usual definition from GRA and KAT. We prove that the assertions (assumptions) form a Boolean algebra. Moreover, we also prove that the assertions defined in the algebra are exactly the program assertions in the model of monotonic Boolean transformers. Having the dual operator and the demonic choice operator we automatically obtain also the angelic choice operator. In [20,14] the enabledness and termination operators are introduced using axioms for DRA and GRA respectively. These operators can be defined in our algebra, and their axioms can be proved as theorems.

In DRA [22], a pre-post specification statement is introduced and it is used to prove that a program refines a pre-post specification statement if and only if the program is correct with respect to the pre and post conditions. The proof of this fact requires the assumption that all programs are conjunctive, fact which does not hold for arbitrary monotonic predicate transformers. We have introduced another specification statement, and we proved a similar result in the absence of the conjunctivity property.

The paper is structured as follows. Section 2 introduces the monotonic Boolean transformers that are the model of our algebra. Section 3 introduces the monotonic Boolean transformers algebra and some of its properties. Section 4 introduces the assertions and the assumptions. Some of their properties are also introduced, and proofs that they form Boolean algebras are given. In Section 5, we define the weakest precondition, the guard of a program, Hoare triples [11] for total correctness, data refinement of programs, and we prove some properties of these constructs. The weakest precondition of top satisfies all axioms set for the termination operator in [20,14], and dually the guard of a program satisfies all axioms set for the enabledness operator in [20,14].

All our results were mechanically verified in the Isabelle [16] theorem prover.

2 Monotonic Boolean Transformers

In this section we introduce the concept of monotonic Boolean transformers which is more general than monotonic predicate transformers. For a set of states X, monotonic predicate transformers over X are monotonic functions from $\mathsf{Pred}.X$ to $\mathsf{Pred}.X$ where $\mathsf{Pred}.X = X \to \mathsf{Bool}$ and Bool is the complete Boolean algebra with two elements, true and false. Monotonic predicate transformers are used for modeling imperative programs. A program is modeled by a predicate transformer S, where if $q \in \mathsf{Pred}.X$ is a predicate (set) of final states, then $S.q$

are the initial states from which the program terminates and if it terminates in a state s, then s is from q.

In this context we only need the assumption that we work with a complete Boolean algebra instead of $\mathsf{Pred}.X$. This generalization is mainly used here because it is sufficient for expressing and proving the properties from this paper. However, we can also apply these results directly to Boolean algebras of the form $X \to Y \to \mathsf{Bool}$, which were used in [3,17,18] for modeling procedures with parameters. Let $\langle B, \wedge, \vee, \leq, \neg, \top, \bot \rangle$ be a complete Boolean algebra. We denote by $\mathsf{Mtran}.B$ the set of all monotonic functions from B to B.

$$\mathsf{Mtran}.B = \{S : B \to B \mid \forall p, q : p \leq q \Rightarrow S.p \leq S.q)\}$$

The elements of $\mathsf{Mtran}.B$ are called monotonic Boolean transformers, or just monotonic transformers, or programs.

We point-wise extend all operations except the negation from B to $\mathsf{Mtran}.B$:

$$\begin{array}{lll} (S \sqcap T).p = S.p \wedge T.p & \mathsf{magic}.p = \top & S \sqsubseteq T = (\forall p : S.p \leq T.p) \\ (S \sqcup T).p = S.p \vee T.p & \mathsf{fail}.p \quad = \bot & \end{array}$$

The extended constants magic and fail are monotonic Boolean transformers and, if S and T are monotonic Boolean transformers, then $S \sqcap T$, and $S \sqcup T$ are monotonic Boolean transformers. We could extend the negation similarly, however the negation applied to a monotonic function is not monotonic.

The program $S \sqcap T$ models the demonic choice between executing S or T. The choice is demonic because the user does not control it. In order for this choice to be correct, both S and T must be correct. The program $S \sqcup T$ models the angelic choice. Here the choice is angelic because the user can decide between executing S or T. This choice is correct if one of the programs S and T is correct. The relation $\sqsubseteq$ is the refinement relation. A program S is refined by a program T if we can replace S by T. The program magic is always correct, but it cannot be implemented. The program fail never terminates. fail is equivalent to while true do skip.

If $S, T \in \mathsf{Mtran}$, $p, q \in \mathsf{Bool}$, then we introduce the transformers $S \circ T$, $\{p\}$, $[p]$, skip, S°, S^ω, $||p|| \in \mathsf{Mtran}$, the sequential composition of S and T, the assert statement of p, the assume statement of p, the skip statement, the dual of S, the iteration of S, and the post-condition statement of p, respectively. These are given by the following definitions:

$$\begin{array}{lll} (S \circ T).p = S.(T.p) & & \text{(sequential composition)} \\ \{p\}.q & = p \wedge q & \text{(assert statement)} \\ [p].q & = \neg p \vee q & \text{(assume statement)} \\ \mathsf{skip}.p & = p & \text{(skip statement)} \\ S^\circ.p & = \neg S.(\neg p) & \text{(dual of a program)} \\ S^\omega & = \mu X : S \circ X \sqcap \mathsf{skip} & \text{(iteration)} \\ ||p||.q & \begin{cases} \top & \text{if } p \leq q \\ \bot & \text{otherwise} \end{cases} & \text{(postcondition statement)} \end{array}$$

The functional composition of monotonic transformers corresponds to the sequential composition of programs. The assert statement $\{p\}$ executed from a state in which the predicate p is true behaves as skip, otherwise fails. The assume statement $[p]$ executed from a state in which the predicate p is true behaves as skip, otherwise behaves as magic. The statement skip does not change the state of computation. The dual was used in [9,4,5,6] for predicate transformers. The term *conjugate* has also been used to name the dual operator. We will use the dual to define the negation of an assertion in the algebra of monotonic predicate transformers. The iteration is used to define the while program:

$$\mathsf{while}\ b\ \mathsf{do}\ S = ([b] \circ S)^{\omega} \circ [\neg b]$$

The conditional program can be introduced using the assert statement and the angelic choice or using the assume statement and the demonic choice:

$$\mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ T = \{b\} \circ S \sqcup \{\neg b\} \circ T = [b] \circ S \sqcap [\neg b] \circ T$$

If $S \in \mathsf{Mtran}$ and $p, q \in B$, then a Hoare triple $p \{\!|S|\!\} q$ is true if $p \leq S.q$.

The post-condition statement $||p||$ has been used in [17,18] to connect total correctness Hoare triples to refinement statements. The following relation is true

$$p \{\!|S|\!\} q \;\Leftrightarrow\; \{p\} \circ ||q|| \sqsubseteq S \tag{1}$$

and it is a consequence of the following theorem.

Theorem 1. *If $p \in B$ then*

1. $||p|| \in \mathsf{Mtran}$ ($||p||$ is monotonic)
2. $||p||.p = \top$
3. $\{S.p\} \circ ||p|| \sqsubseteq S$

Definition 1. *A monotonic transformer S is* disjunctive *if for all $p, q \in B$, $S.(p \vee q) = S.p \vee S.q$, which is equivalent to*

$$(\forall U, V \in \mathsf{Mtran} : S \circ (U \sqcup V) = (S \circ U) \sqcup (S \circ V))$$

A monotonic transformer S is conjunctive *if for all $p, q \in B$, $S.(p \wedge q) = S.p \wedge S.q$, which is equivalent to*

$$(\forall U, V \in \mathsf{Mtran} : S \circ (U \sqcap V) = (S \circ U) \sqcap (S \circ V))$$

Next theorem gives a characterization of assertion statements.

Theorem 2. *A monotonic transformer S is an assertion if and only if $S \sqsubseteq \mathsf{skip}$ and S is disjunctive.*

Proof. If S is an assertion it is easy to prove that $S \sqsubseteq \mathsf{skip}$ and S is disjunctive. Conversely, assume that $S \sqsubseteq \mathsf{skip}$ and S is disjunctive. We prove that $S = \{S.\top\}$.

$$
\begin{array}{ll}
\bullet & \{S.\top\}.q \\
= & \{\text{Boolean algebra property}\} \\
 & \{S.(q \vee \neg q)\}.q \\
= & \{S \text{ is disjunctive}\} \\
 & \{S.q \vee S.(\neg q)\}.q \\
= & \{\text{Definition of assert and distributivity}\} \\
 & (S.q \wedge q) \vee (S.(\neg q) \wedge q) \\
= & \{S \leq \mathsf{skip} \text{ implies} S.q \leq q\} \\
 & S.q \vee (S.(\neg q) \wedge q) \\
= & \{\text{Prove } S.(\neg q) \wedge q = \bot\} \\
 & \begin{array}{ll}
 \bullet & S.(\neg q) \wedge q = \bot \\
 \Leftrightarrow & \{\bot \text{ is the least element}\} \\
 & S.(\neg q) \wedge q \leq \bot \\
 \Leftarrow & \{S \leq \mathsf{skip} \text{ implies} S.(\neg q) \leq \neg q\} \\
 & \neg q \wedge q \leq \bot \\
 \Leftrightarrow & \{\text{Boolean algebra properties}\} \\
 & \text{true}
 \end{array} \\
 & S.q \vee \bot \\
= & \{\text{Boolean algebra properties}\} \\
 & S.q
\end{array}
$$

□

Theorem 3. *A monotonic transformer S is an assumption if and only if $S \geq$ skip and S is conjunctive.*

3 Algebra of Monotonic Boolean Transformers

We introduce in this section an algebraic structure which has as a model the monotonic Boolean transformers.

Definition 2. *An* algebra of monotonic Boolean transformers *(abbreviated* MBT*) is an algebra $\mathcal{A} = \langle A, \sqcap, \sqcup, \circ, \_^{\circ}, \_^{\omega}, 1, \bot, \top\rangle$ where $\sqcap$, $\sqcup$, and $\circ$ are binary operations, $\_^{\circ}$, $\_^{\omega}$ are unary operation and 1, $\bot$, and $\top$ are constants, which satisfies the following axioms:*

(A1) $\langle A, \sqcap, \sqcup, \bot, \top\rangle$ is a bounded distributive lattice
(A2) $\langle A, \circ, 1\rangle$ is a monoid

$(A3)$	$(x \sqcap y) \circ z = (x \circ z) \sqcap (y \circ z)$	$(A8)$	$(x \circ y)^{\circ} = x^{\circ} \circ y^{\circ}$
$(A4)$	$x \leq y \Rightarrow z \circ x \leq z \circ y$	$(A9)$	$(x \circ \top) \sqcap (x^{\circ} \circ \bot) = \bot$
$(A5)$	$\top \circ x = \top$	$(A10)$	$x^{\omega} = x \circ x^{\omega} \sqcap 1$
$(A6)$	$x \leq y \Leftrightarrow y^{\circ} \leq x^{\circ}$	$(A11)$	$x \circ z \sqcap y \leq z \Rightarrow x^{\omega} \circ y \leq z$
$(A7)$	$x^{\circ\circ} = x$		

The algebra of monotonic Boolean transformers includes all operators and axioms of the general refinement algebra introduced in [22], except the weak iteration operator ($\_^*$) and its axioms. Additionally it includes the angelic choice and the dual operator and their corresponding axioms. We also assume that the lattice of the choice operations is distributive. The iteration operator (ω) is introduced here only for completion. It will not be used further in this paper. All properties proved in [22] for ω in the general refinement algebra hold also for MBT algebra.

The dual operator behaves like a negation operator: it is anti-monotonic, it is an involution ($x^{\circ\circ} = x$), and the conjunction of $x \circ \top$ and $x^\circ \circ \bot$ is $\bot$. However, the dual operator applied to a monotonic Boolean transformer is also monotonic. This operator will be used to define the negation for assert and assume statements of MBT algebra.

Alternatively we could introduce only $\sqcap$, $\circ$, $\_^\circ$, $\_^\omega$, 1, and $\top$ as primitive operations, and then define $\sqcup$ and $\bot$ in terms of $\sqcap$, $\_^\circ$, and $\top$.

Theorem 4. *If the constants* 1, $\bot$, *and* $\top$ *from* MBT *are interpreted as* skip, fail, *and* magic, *then the monotonic Boolean transformers are a model for the axioms of* MBT.

Proof. All properties ($A1$) to ($A11$) from Definition 2 are easy to verify.

In [13], multirelations are used to model angelic and demonic nondeterminism, and they are shown to be equivalent to monotonic predicate transformers. This work would enable showing that the multirelations are also a model for MBT algebra.

Next theorem lists a number of properties that are true in a MBT algebra. The properties are direct consequences of the axioms of MBT algebra.

Theorem 5. *In* MBT *the following properties hold:*

1. $\top^\circ = \bot$ and $\bot^\circ = \top$
2. $1^\circ = 1$
3. $(x \sqcap y)^\circ = x^\circ \sqcup y^\circ$
4. $(x \sqcup y)^\circ = x^\circ \sqcap y^\circ$
5. $x = y \Leftrightarrow x^\circ = y^\circ$
6. $(x \sqcup y) \circ z = (x \circ z) \sqcup (y \circ z)$
7. $x \circ (y \sqcap z) \leq (x \circ y) \sqcap (x \circ z)$
8. $x \circ (y \sqcup z) \geq (x \circ y) \sqcup (x \circ z)$
9. $\bot \circ x = \bot$
10. $x \leq y \Rightarrow x \circ z \leq y \circ z$
11. $x \leq y \wedge u \leq v \Rightarrow x \circ u \leq y \circ v$
12. $1 \leq x \Rightarrow y \leq x \circ y$
13. $1 \leq x \Rightarrow y \leq y \circ x$
14. $x \leq 1 \Rightarrow x \circ y \leq y$
15. $x \leq 1 \Rightarrow y \circ x \leq y$
16. $x \leq x \circ \top$ and $x \circ \bot \leq x$
17. $(x \circ \top) \sqcup (x^\circ \circ \bot) = \top$
18. $(x \circ \bot) \sqcup (x^\circ \circ \top) = \top$

Definition 3. *An element* x is conjunctive *if it satisfies*

$$(\forall y, z : x \circ (y \sqcap z) = (x \circ y) \sqcap (x \circ z))$$

and dually x *is* disjunctive *if it satisfies*

$$(\forall y, z : x \circ (y \sqcup z) = (x \circ y) \sqcup (x \circ z))$$

The set of conjunctive and disjunctive elements are denoted by Conj *and* Disj, *respectively.*

As pointed out in Section 2, these definitions are equivalent to the definitions of conjunctive and disjunctive functions in the model of monotonic Boolean transformers.

Lemma 1. *For $x \in \mathsf{MBT}$ the following properties hold*

1. $x \in \mathsf{Conj} \Rightarrow x^{\circ} \in \mathsf{Disj}$
2. $x \in \mathsf{Disj} \Rightarrow x^{\circ} \in \mathsf{Conj}$

4 Assertions and Assumptions

This section introduces the set of assertions and assumptions of a MBT algebra. In a Kleene algebra with tests [12], the tests (which are the equivalent to assumptions) are postulated. The tests are elements of a subset of a Kleene algebra and they form a Boolean algebra. In a demonic refinement algebra [22], guards (which are equivalent to assumptions) are the elements that have a complement with respect to $\sqcap$, $\circ$,1, and $\top$. Because our algebra contains the dual operator we are able to introduce the assertions using a conjunction of an inequality and an equality which is logically simpler than the definition from [22]. We prove that the assertions and also the assumptions are Boolean algebras, and moreover, we prove that the assertions and the assumptions from our algebra correspond exactly to the assertions and the assumptions from the monotonic Boolean transformers model.

Definition 4. *In a* MBT *algebra the set of assertions is defined by*

$$\mathsf{Assertion} = \{p : p \le 1 \wedge p = (p \circ \top) \sqcap p^{\circ}\}$$

Lemma 2. *Let $p \in \mathsf{Assertion}$ then*

1. $p^{\circ} = (p^{\circ} \circ \bot) \sqcup p$
2. $p = (p \circ \top) \sqcap 1$ and $p^{\circ} = (p^{\circ} \circ \bot) \sqcup 1$
3. $p, p^{\circ} \in \mathsf{Conj}$ and $p, p^{\circ} \in \mathsf{Disj}$

Proof. We prove only the last property. First we prove $p \in \mathsf{Conj}$, i.e. for all $x, y \in \mathsf{MBT}$, $p \circ (x \sqcap y) = (p \circ x) \sqcap (p \circ y)$:

$$\begin{array}{ll}
\bullet & p \circ (x \sqcap y) \\
= & \{\text{property 2. of this theorem}\} \\
 & ((p \circ \top) \sqcap 1) \circ (x \sqcap y) \\
= & \{\text{axioms of } \mathsf{MBT}\} \\
 & (p \circ \top \circ (x \sqcap y)) \sqcap x \sqcap y \\
= & \{\text{axioms of } \mathsf{MBT}\} \\
 & (p \circ \top) \sqcap x \sqcap y \\
= & \{\text{lattice properties}\} \\
 & ((p \circ \top) \sqcap x) \sqcap ((p \circ \top) \sqcap y) \\
= & \{\text{axioms of } \mathsf{MBT}\} \\
 & (((p \circ \top) \sqcap 1) \circ x) \sqcup (((p \circ \top) \sqcap 1) \circ y) \\
= & \{\text{property 2. of this theorem}\} \\
 & (p \circ x) \sqcap (p \circ y)
\end{array}$$

The property $p \in \mathsf{Disj}$ can be proved similarly, but we also need to use the distributivity of $\sqcap$ over $\sqcup$.

Finally $p^\circ \in \mathsf{Conj}$ and $p^\circ \in \mathsf{Disj}$ follow using Lemma 1. $\square$

The definition of assertions corresponds to assertions in the model of Boolean transformers.

Theorem 6. *In* Mtran *the set* $\mathsf{Assertion}$ *is the set of all assertions,* $\{p\}$*, for* $p \in \mathsf{Bool}$*.*

Proof. We prove that $\mathsf{Assertion} = \{\{p\} \mid p \in \mathsf{Bool}\}$ in Mtran. First if $p \in \mathsf{Bool}$, it is easy to show that $\{p\} \in \mathsf{Assertion}$. Conversely if $x \in \mathsf{Assertion}$, then by Lemma 5 $x \in \mathsf{Disj}$, and using Theorem 2, it follows that $x \in \{\{p\} \mid p \in \mathsf{Bool}\}$. $\square$

Lemma 3. *If* $p, q \in \mathsf{Assertion}$*, then* $p \circ q = p \sqcap q$*.*

Proof. The inequality $p \circ q \leq p \sqcap q$ follows directly from the axioms of MBT.

The second inequality follows from:

$$\begin{array}{ll}
\bullet & p \sqcap q \\
= & \{\mathsf{Assertion}\ \text{definition}\} \\
 & p \circ \top \sqcap p^\circ \sqcap q \circ \top \sqcap q^\circ \\
\leq & \{\text{sub-derivation}\} \\
 & \begin{array}{ll}
 \bullet & p^\circ \leq p^\circ \circ q^\circ \wedge q^\circ \leq p^\circ \circ q^\circ \\
 = & \{\text{Theorem 5}\} \\
 & \mathsf{true} \\
 \bullet & q \leq p^\circ \circ q \circ \top \\
 \Leftarrow & \{\text{transitivity of } \leq\} \\
 & q \leq p^\circ \circ q \wedge p^\circ \circ q \leq p^\circ \circ q \circ \top \\
 = & \{\text{Theorem 5}\} \\
 & \mathsf{true}
 \end{array} \\
 & p \circ \top \sqcap p^\circ \circ q \circ \top \sqcap p^\circ \circ q^\circ \\
= & \{p^\circ \in \mathsf{Conj}\} \\
 & p \circ \top \sqcap p^\circ \circ (q \circ \top \sqcap q^\circ) \\
= & \{q \in \mathsf{Assertion}\} \\
 & p \circ \top \sqcap p^\circ \circ q \\
= & \{\mathsf{MBT}\ \text{axioms}\} \\
 & (p \circ \top \sqcap p^\circ) \circ q \\
= & \{p \in \mathsf{Assertion}\} \\
 & p^\circ \circ q^\circ
\end{array}$$

This concludes the theorem. $\square$

Definition 5. *For an assertion* $p \in \mathsf{Assertion}$ *the* negation *of* p*, denoted* $\neg p$ *is defined by*

$$\neg p = (p^\circ \circ \bot) \sqcap 1$$

Theorem 7. *The assertions are closed under* $\sqcap$*,* $\sqcup$*,* $\neg$*,* 1*, and* $\bot$*.*

Proof. We prove only that $p \sqcap q \in \mathsf{Assertion}$. It is true that $p \sqcap q \le 1$. We prove also that $p \sqcap q = (p \sqcap q) \circ \top \sqcap (p \sqcap q)^\circ$:

- $\bullet$ $p \sqcap q$
- $=$ {Lemma 3}
  $p \circ q$
- $=$ $\{p, q \in \mathsf{Assertion}\}$
  $(p \circ \top \sqcap p^\circ) \circ (q \circ \top \sqcap q^\circ)$
- $=$ {axioms of MBT}
  $p \circ \top \sqcap p^\circ \circ (q \circ \top \sqcap q^\circ)$
- $=$ $\{p^\circ \in \mathsf{Conj}$ by Lemma 2$\}$
  $p \circ \top \sqcap p^\circ \circ q \circ \top \sqcap p^\circ \circ q^\circ$
- $=$ {MBT axioms}
  $(p \circ \top \sqcap p^\circ) \circ q \circ \top \sqcap p^\circ \circ q^\circ$
- $=$ $\{p \in \mathsf{Assertion}\}$
  $p \circ q \circ \top \sqcap p^\circ \circ q^\circ$
- $=$ {MBT axioms}
  $p \circ q \circ \top \sqcap (p \circ q)^\circ$
- $=$ {Lemma 3}
  $(p \sqcap q) \circ \top \sqcap (p \sqcap q)^\circ$. □

Theorem 8. *The structure* $(\mathsf{Assertion}, \sqcap, \sqcup, \neg, \bot, 1)$ *is a Boolean algebra.*

Proof. The structure $(\mathsf{Assertion}, \sqcap, \sqcup, \bot, 1)$ is a bounded distributive lattice by Theorem 7 and by the fact that MBT is a distributive lattice. We need to show also that $\neg$ satisfies the negation axioms: $p \sqcap \neg p = \bot$ and $p \sqcup \neg p = 1$. We only show here $p \sqcup \neg p = 1$.

- $\bullet$ $p \sqcup \neg p$
- $=$ {Definition of $\neg$}
  $p \sqcup (p^\circ \circ \bot \sqcap 1)$
- $=$ {lattice distributivity}
  $(p \sqcup p^\circ \circ \bot) \sqcap (p \sqcup 1)$
- $=$ $\{p \in \mathsf{Assertion}\}$
  $(p \sqcup p^\circ \circ \bot) \sqcap 1$
- $=$ {Theorem 5 and MBT axioms}
  $(p^\circ \sqcap p \circ \top)^\circ \sqcap 1$
- $=$ $\{p \in \mathsf{Assertion}\}$
  $p^\circ \sqcap 1$
- $=$ $\{p \le 1 \Leftrightarrow 1 \le p^\circ$ by MBT axioms$\}$
  1 □

Next lemma introduces some additional properties for assertions.

Lemma 4. *If* $p \in \mathsf{Assertion}$ *and* $x, y \in \mathsf{MBT}$, *then*

1. $p \circ p = p$ and $p^\circ \circ p^\circ = p^\circ$
2. $p \circ p^\circ = p$ and $p^\circ \circ p = p^\circ$

3. $p^\circ \circ x \sqcup (\neg p) \circ \top = p^\circ \circ x$
4. $p \circ x \sqcup (\neg p) \circ y = p^\circ \circ x \sqcap (\neg p)^o \circ y$

Proof. We prove only the last property:

$$
\begin{array}{ll}
\bullet & p \circ x \sqcup (\neg p) \circ y \\
= & \{p, \neg p \in \mathsf{Assertion}\} \\
 & (p \circ \top \sqcap p^\circ) \circ x \sqcup ((\neg p) \circ \top \sqcap (\neg p)^\circ) \circ y \\
= & \{\mathsf{MBT}\text{ axioms}\} \\
 & (p \circ \top \sqcap p^\circ \circ x) \sqcup ((\neg p) \circ \top \sqcap (\neg p)^\circ \circ y) \\
= & \{\text{lattice distributivity}\} \\
 & ((p \circ \top \sqcap p^\circ \circ x) \sqcup (\neg p) \circ \top) \sqcap ((p \circ \top \sqcap p^\circ \circ x) \sqcup (\neg p)^\circ \circ y) \\
= & \{\text{lattice distributivity}\} \\
 & (p \circ \top \sqcup (\neg p) \circ \top) \sqcap (p^\circ \circ x \sqcup (\neg p) \circ \top) \sqcap (p \circ \top \sqcup (\neg p)^\circ \circ y) \sqcap (p^\circ \circ x \sqcup (\neg p)^\circ \circ y) \\
= & \{\mathsf{MBT}\text{ axioms}\} \\
 & (p \sqcup (\neg p)) \circ \top \sqcap (p^\circ \circ x \sqcup (\neg p) \circ \top) \sqcap (p \circ \top \sqcup (\neg p)^\circ \circ y) \sqcap (p^\circ \circ x \sqcup (\neg p)^\circ \circ y) \\
= & \{\mathsf{Assertion}\text{ is a Boolean algebra}\} \\
 & (p^\circ \circ x \sqcup (\neg p) \circ \top) \sqcap (p \circ \top \sqcup (\neg p)^\circ \circ y) \sqcap (p^\circ \circ x \sqcup (\neg p)^\circ \circ y \\
= & \{\text{Property 3: } p^\circ \circ x \sqcup (\neg p) \circ \top = p^\circ \circ x\} \\
 & p^\circ \circ x \sqcap (\neg p)^\circ \circ y \sqcap (p^\circ \circ x \sqcup (\neg p)^\circ \circ y) \\
= & \{\text{lattice properties}\} \\
 & p^\circ \circ x \sqcap (\neg p)^\circ \circ y
\end{array}
$$

□

The property 3. from Lemma 4 shows that the two ways of defining the conditional program are also equivalent in MBT. In MBT the conditional program is defined by

$$\mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ y = b \circ x \sqcup \neg b \circ y = b^\circ \circ x \sqcap (\neg b)^\circ \circ y$$

The assumptions of MBT are defined similarly to assertions, but using the duals of the properties for assertions.

Definition 6. *The assumptions of* MBT, *denoted by* Assumption $\subseteq$ MBT, *are defined by*

$$\mathsf{Assumption} = \{g : 1 \leq g \wedge g = (g \circ \bot) \sqcup g^\circ\}$$

Lemma 5. *Let* $g \in$ Assumption *then*

1. $g \in \mathsf{Assumption} \Leftrightarrow g^\circ \in \mathsf{Assertion}$
2. $g^\circ = (g^\circ \circ \top) \sqcap g$
3. $g = (g \circ \bot) \sqcup 1$ and $g^\circ = (g^\circ \circ \top) \sqcap 1$
4. $g, g^\circ \in \mathsf{Conj}$ and $g, g^\circ \in \mathsf{Disj}$

Proof. These properties can be proved similarly to those for assertion, but using the dual properties. □

Theorem 9. *In* Mtran *the* Assumption *is the set of all assumptions,* $[p]$, *for* $p \in B$.

Proof. This fact can be proved similarly to Theorem 6, using Theorem 3. This theorem can also be proved using Lemma 5.1, and the fact that in Mtran $\{p\}^\circ = [p]$. $\square$

The negation of an assumption can be defined using the negation of an assertion.

Definition 7. *The negation of an assumption* $g \in \mathsf{Assumption}$*, denoted* $\neg g \in \mathsf{Assumption}$*, is given by*

$$\neg q = (\neg g^\circ)^\circ$$

Theorem 10. *The assumptions are closed to the operations* $\sqcap$*,* $\sqcup$*,* $\neg$*,* 1*, and* $\top$*, and the structure* $(\mathsf{Assumption}, \sqcap, \sqcup, \neg, 1, \top)$ *is a Boolean algebra.*

5 Weakest Precondition, Guards, Hoare Triples, and Data Refinement

This sections introduces the weakest precondition for elements of a MBT algebra, and using it introduces valid Hoare triples. Various results connecting valid Hoare triples, refinement, and data refinement are also proved.

Definition 8. *The* weakest precondition *of a program* x *and* $\top$*, denoted* $\mathsf{wpt}.x \in \mathsf{MBT}$*, is given by*

$$\mathsf{wpt}.x = (x \circ \top) \sqcap 1.$$

This definition is justified by the fact that in the monotonic Boolean transformer model $\mathsf{wpt}.S$ is equal to $\{S.\top\}$ and $\mathsf{wpt}.(S \circ \{p\})$ is equal to $\{S.p\}$. The operator wpt satisfies all axioms set for the termination operator in [20,14]. These axioms are listed among the conclusions of the next theorem.

Theorem 11. *The following properties are true for* wpt.

1. $\mathsf{wpt}.x \in \mathsf{Assertion}$
2. $(\mathsf{wpt}.x) \circ x = x$
3. $p \in \mathsf{Assertion} \Rightarrow \mathsf{wpt}.p = p$
4. $p \in \mathsf{Assertion} \wedge p \circ x = x \Rightarrow \mathsf{wpt}.x \leq p$
5. $p \in \mathsf{Assertion} \Rightarrow \mathsf{wpt}.(p^\circ) = 1$
6. $p, q \in \mathsf{Assertion} \Rightarrow \mathsf{wpt}.(p^\circ \circ q) = \neg p \sqcup q$
7. $x \leq y \Rightarrow \mathsf{wpt}.x \leq \mathsf{wpt}.y$
8. $\mathsf{wpt}.(x \circ y) = \mathsf{wpt}.(x \circ \mathsf{wpt}.y)$
9. $p \in \mathsf{Assertion} \wedge x \in \mathsf{Conj} \Rightarrow x \circ p = \mathsf{wpt}.(x \circ p) \circ x$ (moving assertions)
10. $(\mathsf{wpt}.x) \circ \top = x \circ \top$

Proof. We only show here the proof of property 6.

$\bullet$ $\quad \mathsf{wpt}.(p^\circ \circ q)$

$=$ $\quad \{\text{Definition}\}$

$\quad (p^\circ \circ q \circ \top) \sqcap 1$

$=$ {Lemma 2}
$(((p^\circ \circ \bot) \sqcup 1) \circ q) \sqcap 1$
$=$ {MBT axioms}
$((p^\circ \circ \bot) \sqcup q) \sqcap 1$
$=$ {lattice distributivity}
$((p^\circ \circ \bot) \sqcap 1) \sqcup (q \sqcap 1)$
$=$ $\{q \in \mathsf{Assertion}\}$
$((p^\circ \circ \bot) \sqcap 1) \sqcup q$
$=$ {definition of $\neg$}
$\neg p \cup q$ □

In Mtran the guard of a program is defined as the set of all states from which the program is guaranteed to terminate. Formally in Mtran the guard of a program S is the predicate $\neg S.\bot$. In MBT we can also define the guard of a program as an assumption.

Definition 9. *The guard of an element $x \in \mathsf{MBT}$, denoted $\mathsf{grd}.x$, is given by*

$$\mathsf{grd}.x = x \circ \bot \sqcup 1.$$

In Mtran the guard of a program S corresponds to $[\neg S.\bot]$: $\mathsf{grd}.S = [\neg S.\bot]$

The operator grd satisfies all axioms set for the termination operator in [20,14]. These axioms are listed among the conclusions of the next theorem.

Theorem 12. *If $x \in \mathsf{MBT}$, and $p \in \mathsf{Assertion}$, then*

1. $\mathsf{grd}.x \in \mathsf{Assumption}$
2. $\mathsf{grd}.x \circ x = x$
3. $\mathsf{grd}.x = (\neg \mathsf{wpt}.(x \circ \bot))^\circ$
4. $g \in \mathsf{Assumption} \Rightarrow g \leq \mathsf{grd}.(g \circ x)$
5. $\mathsf{grd}.(x \circ y) = \mathsf{grd}.(x \circ \mathsf{grd}.y)$
6. $(\mathsf{grd}.x) \circ \bot = x \circ \bot$

Proof. We prove only the property 3 here:

$\bullet$ $(\neg \mathsf{wpt}.(x \circ \bot))^\circ$
$=$ {definition of $\neg$}
$((\mathsf{wpt}.(x \circ \bot))^\circ \circ \bot \sqcap 1)^\circ$
$=$ {Theorem 5 and MBT axioms}
$(\mathsf{wpt}.(x \circ \bot)) \circ \top \sqcup 1$
$=$ {definition of wpt}
$(x \circ \bot \circ \top \sqcap 1) \circ \top \sqcup 1$
$=$ {Theorem 5 and MBT axioms}
$x \circ \bot \sqcup 1$
$=$ {definition of grd}
$\mathsf{grd}.x.$ □

Definition 10. *For $p, q, x \in \mathsf{MBT}$, the* Hoare total correctness triple $p \{\!|x|\!\} q \in$ Bool *is defined by*

$$(p \{\!|x|\!\} q) := p \leq \mathsf{wpt}.(x \circ q).$$

This definition also corresponds to the classical definition of Hoare total correctness triples in the monotonic Boolean transformers lattice. If $p, q \in \mathsf{Bool}$ and $S \in \mathsf{Mtran}$, then $\{p\}\ \{\!|S|\!\}\ \{q\}$ is equivalent to $p \leq S.q$.

In [22] the total correctness triple of a program x with respect to a precondition p and a post-condition q is defined by $p^\circ \circ x \circ (-q)^\circ = \top$. Next theorems shows that this definition is equivalent to our definition.

Theorem 13. *If* $p \in \mathsf{Assertion}$ *then*

$$p\,\{\!|x|\!\}\,q \;\Leftrightarrow\; p^\circ \circ x \circ (\neg q)^\circ = \top$$

Proof. First assume $p\,\{\!|x|\!\}\,q$, which implies $p \circ \top \leq x \circ q \circ \top$. Show $p^\circ \circ x \circ (\neg q)^\circ = \top$.

$$\begin{array}{ll}
\bullet & \top \\
= & \{\text{Theorem } 5\} \\
 & (x \circ q)^\circ \circ \bot \sqcup x \circ q \circ \top \\
\leq & \{\text{the assumption implies } p \circ \top \leq x \circ q \circ \top\} \\
 & p^\circ \circ \bot \sqcup x \circ q \circ \top \\
= & \{\text{Theorem } 5\} \\
 & (p^\circ \circ \bot \sqcup 1) \circ x \circ q \circ \top \\
= & \{\text{Lemma } 2\} \\
 & p^\circ \circ x \circ q \circ \top \\
\leq & \{\mathsf{MBT} \text{ axioms}\} \\
 & p^\circ \circ x \circ (q \circ \top \sqcup 1) \\
= & \{\text{Theorem } 5\} \\
 & p^\circ \circ x \circ (q^\circ \circ \bot \sqcap 1)^\circ \\
= & \{\text{definition of } \neg\} \\
 & p^\circ \circ x \circ (\neg q)^\circ
\end{array}$$

For the second implication assume $p^\circ \circ x \circ (\neg q)^\circ = \top$, which is equivalent to $p^\circ \circ x \circ (q \circ \top \sqcup 1) = \top$. To show $p\,\{\!|x|\!\}\,q$ it is enough to show $p \leq x \circ q \circ \top$, which follows from $p \leq p \circ \top = p \circ x \circ q \circ \top \leq x \circ q \circ \top$.

$$\begin{array}{ll}
\bullet & p \circ \top \\
= & \{\mathsf{MBT} \text{ axioms}\} \\
 & p \circ \top \circ \bot \\
= & \{\text{assumption}\} \\
 & p \circ p^\circ \circ x \circ (q \circ \top \sqcup 1) \circ \bot \\
= & \{\text{Lemma } 4\} \\
 & p \circ x \circ (q \circ \top \sqcup 1) \circ \bot \\
= & \{\text{Theorem } 5\} \\
 & p \circ x \circ q \circ \top
\end{array}$$

□

Definition 11. *For* $x, y, u, v \in \mathsf{MBT}$*, the program* x *is* data refined *by the program* y *via the programs* u *and* v*, denoted* $x \sqsubseteq_{u,v} y$*, if*

$$u \circ x \leq y \circ v.$$

This definition for data refinement was used in [19] for constructing invariant based programs using data refinement. Next theorem allows to conclude a correctness statement for a program y which data refines a program x, knowing that x is correct.

Theorem 14. *If* $p, x, y, q, u, v \in \mathsf{MBT}$*, then*

1. $p \{\!|x|\!\} q \wedge x \sqsubseteq_{u,v} y \Rightarrow \mathsf{wpt}.(u \circ p) \{\!|y|\!\} \mathsf{wpt}.(v \circ q)$
2. $p \in \mathsf{Assertion} \wedge p \{\!|x|\!\} q \wedge p \circ x \sqsubseteq_{u,v} y \Rightarrow \mathsf{wpt}.(u \circ p) \{\!|y|\!\} \mathsf{wpt}.(w \circ q)$

Proof. We prove only the second property. The first one has a similar proof. Assume $p \{\!|x|\!\} q$ ($\Leftrightarrow p \le \mathsf{wpt}.(x \circ q)$) and $p \circ x \sqsubseteq_{u,v} y$ ($\Leftrightarrow u \circ p \circ x \le y \circ v$).

$$
\begin{array}{ll}
\bullet & \mathsf{wpt}.(u \circ p) \{\!|y|\!\} \mathsf{wpt}.(v \circ q) \\
= & \{\text{definition of Hoare triple}\} \\
& \mathsf{wpt}.(u \circ p) \le \mathsf{wpt}.(y \circ \mathsf{wpt}.(v \circ q)) \\
= & \{\text{Theorem 11}\} \\
& \mathsf{wpt}.(u \circ p) \le \mathsf{wpt}.(y \circ v \circ q) \\
\Leftarrow & \{\text{assumption and } \mathsf{wpt} \text{ monotonic}\} \\
& \mathsf{wpt}.(u \circ p) \le \mathsf{wpt}.(u \circ p \circ x \circ q) \\
= & \{\text{Theorem 11}\} \\
& \mathsf{wpt}.(u \circ p) \le \mathsf{wpt}.(u \circ p \circ \mathsf{wpt}.(x \circ q)) \\
\Leftarrow & \{\text{assumption and } \mathsf{wpt} \text{ monotonic}\} \\
& \mathsf{wpt}.(u \circ p) \le \mathsf{wpt}.(u \circ p \circ p) \\
= & \{\text{Lemma 4}\} \\
& \mathsf{wpt}.(u \circ p) \le \mathsf{wpt}.(u \circ p) \\
= & \{\le \text{ is reflexive}\} \\
& \mathsf{true}
\end{array}
$$
□

The second property of Theorem 14 is preferable to the first one because the data refinement $p \circ x \sqsubseteq_{u,v} y$ is easier to prove compared to $x \sqsubseteq_{u,v} y$. In $p \circ x \sqsubseteq_{u,v} y$ the properties from p can be used as assumption in the proof.

In [22], von Wright uses a statement called havoc to introduce a pre-post-condition specification statement. von Wright proves that the specification statement is refined by another program x if and only if x is totally correct with respect to the pre and post conditions. However the proof from [22] uses the property that all programs are conjunctive, which does not hold in our setting. We introduce another concept that can be used to define the specification statement and we can prove the equivalence between the refinement of the specification statement into x and the correctness statement of x. As in case of [22], this concept cannot be defined and we use two axioms for introducing it. We assume that we have a function $|\_| : \mathsf{Assertion} \to \mathsf{MBT}$ that satisfies the additional axioms:

$$(P1) \quad |p| \circ p \circ \top = \top \qquad (P2) \quad x \circ p \circ \top \sqcap |p| \le x$$

In the model of monotonic Boolean transformers, if we define $|\{p\}| = ||p||$, then the axioms $(P1)$ and $(P2)$ are satisfied.

Theorem 15. *If* $p, q \in \mathsf{Assertion}$ *and* $x \in \mathsf{MBT}$*, then* $p \{\!|x|\!\} q \Leftrightarrow p \circ |q| \le x$*.*

Proof. Assume $p \, \{\!|x|\!\} \, q$ which is equivalent to $p \leq x \circ q \circ \top$.

$$
\begin{array}{ll}
\bullet & p \circ |q| \\
= & \{\text{Lemma 2}\} \\
 & (p \circ \top \sqcap 1) \circ |q| \\
= & \{\mathsf{MBT}\text{ axioms}\} \\
 & p \circ \top \sqcap |q| \\
\leq & \{\text{assumption}\} \\
 & x \circ q \circ \top \sqcap |q| \\
\leq & \{\text{axiom } (P2)\} \\
 & x
\end{array}
$$

Conversely assume $p \circ |q| \leq x$ and show $p \leq x \circ q \circ \top$ which is equivalent to $p \, \{\!|x|\!\} \, q$ when $p \in \mathsf{Assertion}$

$$
\begin{array}{ll}
\bullet & p \\
= & \{\text{Lemma 2}\} \\
 & (p \circ \top \sqcap 1) \\
= & \{\text{axiom } (P1)\} \\
 & p \circ |q| \circ q \circ \top \sqcap 1 \\
\leq & \{\text{lattice properties}\} \\
 & p \circ |q| \circ q \circ \top \\
\leq & \{\text{assumption}\} \\
 & x \circ q \circ \top
\end{array}
$$

$\square$

6 Conclusions

We have introduced a new algebra for reasoning about imperative programming languages which supports total correctness, refinement, data refinement, demonic choice, and angelic choice. Compared to earlier versions of program algebras, this approach uses the dual of a program as a primitive operation, and the assertion statements are defined using weaker properties than how they were defined in previous work.

We proved a number of results about assertions and assumptions. We have also proved two main theorems. One theorem states a result which can be used to prove the correctness of a concrete program y, by proving that y data refines a program x and x is correct. The other theorem shows the equivalence between the refinement of a specification statement and a Hoare total correctness triple.

All results presented in this paper were mechanically verified in the Isabelle theorem prover.

References

1. Back, R.-J.: On the correctness of refinement in program development. PhD thesis, Department of Computer Science, University of Helsinki (1978)

2. Back, R.-J.: Correctness preserving program refinements: proof theory and applications. Mathematical Centre Tracts, vol. 131. Mathematisch Centrum, Amsterdam (1980)
3. Back, R.-J., Preoteasa, V.: An algebraic treatment of procedure refinement to support mechanical verification. Formal Aspects of Computing 17, 69–90 (2005)
4. Back, R.-J., von Wright, J.: A lattice-theoretical basis for a specification language. In: van de Snepscheut, J.L.A. (ed.) MPC 1989. LNCS, vol. 375, pp. 139–156. Springer, Heidelberg (1989)
5. Back, R.-J., von Wright, J.: Duality in specification languages: a lattice-theoretical approach. Acta Inf. 27, 583–625 (1990)
6. Back, R.-J., von Wright, J.: Refinement Calculus. A systematic Introduction. Springer, Heidelberg (1998)
7. Dang, H.-H., Höfner, P., Möller, B.: Algebraic separation logic. Journal of Logic and Algebraic Programming 80(6), 221–247 (2011); Relations and Kleene Algebras in Computer Science
8. Desharnais, J., Möller, B., Struth, G.: Kleene algebra with domain. ACM Trans. Comput. Logic 7, 798–833 (2006)
9. Guerreiro, P.: Another characterization of weakest preconditions. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 164–177. Springer, Heidelberg (1982), doi:10.1007/3-540-11494-7_12
10. Hoare, C.A., Möller, B., Struth, G., Wehrman, I.: Concurrent kleene algebra. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 399–414. Springer, Heidelberg (2009)
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
12. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. 19, 427–443 (1997)
13. Martin, C.E., Curtis, S.A., Rewitzky, I.: Modelling angelic and demonic nondeterminism with multirelations. Science of Computer Programming 65(2), 140–158 (2007); Special Issue dedicated to selected papers from the conference of program construction 2004 (MPC 2004)
14. Meinicke, L., Solin, K.: Refinement algebra for probabilistic programs. Formal Aspects of Computing 22, 3–31 (2010), doi:10.1007/s00165-009-0111-1
15. Morgan, C.: Programming from specifications. Prentice-Hall, Inc., Englewood Cliffs (1990)
16. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
17. Preoteasa, V.: Program Variables – The Core of Mechanical Reasoning about Imperative Programs. PhD thesis, Turku Centre for Computer Science (November 2006)
18. Preoteasa, V.: Frame rule for mutually recursive procedures manipulating pointers. Theoretical Computer Science 410(42), 4216–4233 (2009)
19. Preoteasa, V., Back, R.-J.: Data refinement of invariant based programs. Electronic Notes in Theoretical Computer Science 259, 143–163 (2009); Proceedings of the 14th BCS-FACS Refinement Workshop (REFINE 2009)
20. Solin, K., von Wright, J.: Enabledness and termination in refinement algebra. Sci. Comput. Program. 74, 654–668 (2009)
21. von Wright, J.: From kleene algebra to refinement algebra. In: Boiten, E.A., Möller, B. (eds.) MPC 2002. LNCS, vol. 2386, pp. 233–262. Springer, Heidelberg (2002)
22. von Wright, J.: Towards a refinement algebra. Sci. Comput. Program. 51, 23–45 (2004)

Behavioural Preservation in Fault Tolerant Patterns

Diego Machado Dias and Juliano Manabu Iyoda

Centro de Informática,
Universidade Federal de Pernambuco,
Recife - PE, Brazil, CEP 50740-560
{dmd,jmi}@cin.ufpe.br

Abstract. In the development of critical systems it is common practice to make use of redundancy in order to achieve higher levels of reliability. There are well established design patterns that introduce redundancy and that are widely documented and adopted by the industry. However there have been few attempts to formally verify some of them. In this work we modelled three fault tolerant patterns (homogeneous redundancy, heterogeneous redundancy and triple modular redundancy) using the HOL4 theorem prover in order to prove that the application of these patterns preserves the behaviour of the original system. Our model takes into account that the original system (without redundancy) computes a certain function with some delay and is amenable to random failure. We illustrate our approach with a case study that verifies in HOL4 that a fault tolerant design applied to a simplified avionic elevator system does not introduce functional errors. This work has been done in collaboration with the Brazilian aircraft manufacturer Embraer.

Keywords: Redundancy management, fault tolerance, behavioural preservation, theorem proving, HOL.

1 Introduction

Critical systems are developed in such a way that safety is addressed explicitly and under severe regulations. For instance, catastrophic failure of digital flight control systems must be extremely improbable to occur: the failure rate must be lower than 10^{-9} per hour [13]. In order to achieve such restriction, critical systems are replicated in different ways in order to guarantee that a failure on one component is covered by another replica of it [8]. Therefore fault tolerance is mostly based on redundancy.

Redundancy is implemented in different ways. We can replicate a system with an identical copy of it (and add a monitor to check which one is working); or we can make copies of a system that have different design and implementation, but that computes the same function; or we can ask for a voter to output an average value of the output of the replicas; and so on. These design solutions, which we call here fault tolerant patterns, are widely used in industry.

Fault tolerant patterns are *de facto* standards in industry. Unfortunately, there has been few attempts to formally verify this catalogue of design patterns. Moreover, the application of them in practice usually results in minor variations of the patterns. In this paper, we model in HOL4 [9] three fault tolerant patterns used in industry and prove that the application of these patterns preserves the behaviour of the original system. Our correctness theorems are compositional, allowing us to prove the correctness of a composition of fault tolerant patterns. Currently, the proofs still need be user-guided, but it is not difficult to fully automate that. This work is the first step towards a general framework for engineers to design and prove the correctness of their variations of the patterns.

Pioneering work on the verification of fault tolerant patterns was done in the nineties [4,16,18]. These works proved the correctness of fault tolerant patterns by hand [4], by using theorems provers [16], and with model checkers [18]. More recently, Dajani-Brown *et al.* [7] used SCADE's [1] model checker to verify the correctness of a triple redundant system. Our work differs from previous works on the compositionality of the theorems and on the separation of concerns: the failure rate and the functional behaviour of the system are separate entities. We do not assume any specific failure rate of the system in order to prove the behavioural preservation (contrary to previous works).

This work is concerned with the non introduction of design error by redundancy. We are not concerned with the benefits of redundancy with respect to failure rate improvements. There are others studies [2][14] that show the quantitative benefits of using redundancy and discuss aspects as cost increase and modifiability analysis.

We illustrate our approach with a case study in which we apply a triple modular redundancy pattern to a highly simplified model of an avionics elevator. We show that it is easy to prove that the addition of the redundancy did not introduce more bugs to the original (non-replicated) system. This work has been done in collaboration with the Brazilian aircraft manufacturer Embraer.

This paper is organised as follows. Section 2 briefly introduces higher order logic in HOL4. Section 3 describes how we model the fault tolerant patterns in HOL4 and show their correctness theorems. Section 4 illustrates our approach in a case study. Section 5 presents the related work in more details and Section 6 concludes.

2 Overview of HOL

This section briefly introduces the HOL logic and one of its mechanisation, the HOL4 system [11]. HOL is a predicate calculus with typed λ-calculus terms. The predicate calculus of HOL allows variables to range over functions and predicates. For example, Peano's *Mathematical Induction* postulate is naturally formalised in HOL.

$$\vdash \ \forall P.\ (P\ 0)\ \wedge\ (\forall n.\ P\ n \Rightarrow\ P\ (\mathsf{SUC}\ n))\ \Rightarrow\ (\forall n.\ P\ n)$$

The variable P ranges over predicates. If P holds for 0 and if whenever it holds for a number n, it also holds for its successor ($\mathsf{SUC}\ n$), then P holds for all natural numbers.

Function applications in HOL have the form $(M\ N)$. Functions can take functions as arguments and return functions as results and can also be partially applied to an argument. For example, the function $(add\ n\ m = n{+}m)$ takes an argument $n = 3$ and returns a function that takes a number m and adds 3 to it.

HOL is a typed logic. The version of higher-order logic presented here extends Church's simple type theory [6] with polymorphic types. For example, the equality operator $=$ is a higher-order function of type $\alpha \rightarrow (\alpha \rightarrow bool)$. The type of its arguments is not defined *a priori*. The type variables σ (and α, β, etc.) denote 'any type'. The type $\sigma_1 \rightarrow \sigma_2$ denotes the set of all total functions from values of σ_1 to values of σ_2. In the Peano's Mathematical Induction postulate presented above, the predicate P is of type $num \rightarrow bool$, where num is the type of natural numbers. We can write $P : num \rightarrow bool$ to explicitly declare its type. In particular, the HOL4 system provides the *option* type operator in order to 'lift' a type α to a new type containing all values of α plus a special value called $NONE$. For example, the type *num option* contains all natural numbers and $NONE$. In order to distinguish *num* values from $NONE$, numbers are built with the constructor $SOME$: $(SOME\ \mathit{0})$, $(SOME\ \mathit{1})$, $(SOME\ \mathit{2})$, etc. Values from the original type can be recovered by using the function $THE : \alpha\ option \rightarrow \alpha$. For instance, $THE(SOME\ \mathit{3}) = \mathit{3}$. We use *option* types to model input and output signals that may come from a (temporarily) broken machine, i.e. a machine that occasionally communicates an invalid value or no value at all.

The system we use in this work is the HOL4 [11]. The HOL4 system is the latest version of a series of implementations first released in 1988. The HOL4 system was the first mechanisation of higher-order logic and was originally developed for hardware verification [9].

3 Fault Tolerant Patterns

Developing a fault tolerant system is an exercise in exploiting and managing *redundancy*, that is the property of having more of a resource than is minimally necessary to perform the job [14]. The objective for using fault tolerance is to increase the reliability of a system, which is achieved by maintaining some form of correctness despite the presence of faults.

Fault tolerance addresses different classes of failures like systematic failures and random failures depending on the type of redundancy applied. A systematic failure is a flaw on the design of the system (i.e. a bug, in the software engineering terminology). A random failure is a failure caused by wear, deterioration, fatigue, etc.

Redundancy comes in different flavours. We can duplicate a system, triplicate it, decide which replica is the "winner" (or is correct) by voting, install the replicas on top of a reliable architecture, duplicate a system by another that is slightly different from the original one (it may implement a different algorithm or use a different technology), and so on. There are plenty of such solutions that we refer to here as *fault tolerant patterns*.

We call the original system (the target of our replication) a *channel*. A channel is an end-to-end system that goes all the way from acquisition of relevant data (the input) to the generation of the output based on that data [8].

Before introducing the fault tolerant patterns used in this work, we introduce the formal model of the specification and the formal model of the components that make up the implementation. Later on we formally define 3 patterns: homogeneous redundancy, heterogeneous redundancy and triple modular redundancy. These patterns were chosen to give us a proof of concept of our framework, since they are the most basic patterns.

3.1 Specification

We specify a channel as a black-box called *SYSTEM* with inputs and outputs. Every *SYSTEM* computes a function f with a certain initialisation delay d. Occasionally, the *SYSTEM* may 'break' as a consequence of random failures. We model a random error of a *SYSTEM* as a function e from time to boolean. We assume that the time is discrete with a common time reference. Whenever $e(t)$ is true, it means that the *SYSTEM* presents a random failure at time t. We call e a *fail function*.

$$\begin{array}{l} \mathit{SYSTEM}\ d\ e\ f\ (\mathit{inp}, \mathit{out})\ = \\ \forall t.\ \mathit{out}\ (t+d) = \mathit{if}\ (\mathit{IS\_NONE}(\mathit{inp}\ t) \vee e(t))\ \mathit{then} \\ \qquad\qquad\qquad\qquad\quad \mathit{NONE} \\ \qquad\qquad\qquad\quad \mathit{else}\ \mathit{SOME}(f\ t\ (\mathit{THE}(\mathit{inp}\ t))) \end{array}$$

Signals are modelled as functions from time (natural numbers) to α *option*. Note that we do not restrict data types of the input. A signal whose value is *NONE* at time t means that the signal in time t comes with an error; and a signal whose value is $\mathit{SOME}(v)$ means that the signal has no errors and carries the value v at time t. There are two conditions that make the output signal of a *SYSTEM* to be *NONE*. Either the input comes with an error ($\mathit{IS\_NONE}(\mathit{inp}\ t)$), in which case it is impossible to compute anything from *NONE*, or the *SYSTEM* itself breaks ($e(t)$). If none of these conditions happen, then *SYSTEM* outputs the result of the computation of f applied to $\mathit{THE}(\mathit{inp}\ t)$. The computation f takes as input the time t and $\mathit{THE}(\mathit{inp}\ t)$.

3.2 Implementation

This section introduces the components that are used to describe the implementation of the fault tolerant patterns.

A *DEL* is a polymorphic delay component that introduces a delay d in an input signal.

$$\mathit{DEL}\ d\ (\mathit{inp}, \mathit{out})\ = \forall t.\ \mathit{out}\ (t+d) = (\mathit{inp}\ t)$$

The output *out* at time $t + d$ is equal to the value of *inp* at time t. The delay refers to the initialisation time of the system.

An *ERROR* component introduces a random error in a signal at time t based on the fail function e.

$$ERROR\ e\ (inp, out)\ = \forall t.\ out\ t = if\ \ e(t)\ then\ NONE\ else\ (inp\ t)$$

The *ERROR* component outputs *NONE* whenever $e(t)$ decides that an error should be introduced at time t.

The combinatorial component *COMB* applies a function $f : num \to \alpha \to \beta$ to the input at each instant in time.

$$\begin{aligned} COMB\ f\ (inp, out)\ = \forall t.\ out\ t = if\ & IS\_NONE(inp\ t)\ then \\ & NONE \\ else\ & SOME(f\ t\ (THE(inp\ t))) \end{aligned}$$

If the input signal at time t comes with an error (i.e. the input is *NONE*), then the broken signal is propagated, otherwise f is applied to the input. The function *IS_NONE* is part of the HOL4 *option* type library. It tests if its argument is a *NONE*. Similarly, there is the function *IS_SOME* that tests if its argument is $SOME(v)$.

The *BUS* combines two signals *inp1* and *inp2* into one signal.

$$\begin{aligned} & BUS\ (inp1, inp2, out) = \\ & \forall t.\ out\ t = if\ \ IS\_NONE(inp1\ \ t)\ \wedge\ IS\_NONE(inp2\ \ t)\ then \\ & \qquad\qquad NONE \\ & \qquad\quad else\ SOME(inp1\ \ t, inp2\ \ t) \end{aligned}$$

If, at time t, both signals $(inp1 : num \to \alpha\ option)$ and $(inp2 : num \to \beta\ option)$ are *NONE*, then *BUS* outputs *NONE*. Otherwise it outputs $SOME(inp1\ t, inp2\ t)$. The output is a pair of options whose type is $(\alpha\ option \times \beta\ option)\ option$. Note that the *BUS* outputs *NONE*, $SOME(NONE, SOME(v))$, $SOME(SOME(v), NONE)$, or $SOME(SOME(v1), SOME(v2))$. But it never outputs $SOME(NONE, NONE)$, which is equivalent to NONE, in order to avoid a component that takes it as input to regard it as a valid data.

TBUS is an extension of *BUS* for three input signals.

$$\begin{aligned} & TBUS\ (inp1, inp2, inp3, out) = \\ & \forall t.\ out\ t{=}if\ \ IS\_NONE(inp1\ \ t) \wedge IS\_NONE(inp2\ \ t) \wedge IS\_NONE(inp3\ \ t)\ then \\ & \qquad\qquad NONE \\ & \qquad\quad else\ SOME(inp1\ \ t, inp2\ \ t, inp3\ \ t) \end{aligned}$$

The *MUX* component separates a combined signal of type $(\alpha\ option \times \beta\ option)$ *breakoption* into two signals. When *inp t* is *NONE*, it outputs (*NONE*, *NONE*). Otherwise, the input at time t has the form $SOME(x, y)$ and the outputs are $out1\ \ t = x$ and $out2\ \ t = y$.

$$
\begin{aligned}
& \mathit{MUX}(\mathit{inp}, (\mathit{out1}, \mathit{out2})) = \\
& \quad \forall t.\ (\mathit{out1}\ t, \mathit{out2}\ t) = \\
& \qquad (\mathit{if}\ \mathit{IS\_NONE}(\mathit{inp}\ t)\ \mathit{then}\ \mathit{NONE} \\
& \qquad\ \mathit{else}\ \mathit{FST}(\mathit{THE}(\mathit{inp}\ t)), \\
& \qquad\ \mathit{if}\ \mathit{IS\_NONE}(\mathit{inp}\ t)\ \mathit{then}\ \mathit{NONE} \\
& \qquad\ \mathit{else}\ \mathit{SND}(\mathit{THE}(\mathit{inp}\ t)))
\end{aligned}
$$

The *MUX* component undoes what the *BUS* component does. The *BUS* takes as input two signals and outputs one signal (made of a pair of values), while the *MUX* takes one signal made of a pair of values and outputs two signals.

TMUX is an extension of *MUX* for three output signals.

$$
\begin{aligned}
& \mathit{TMUX}(\mathit{inp}, (\mathit{out}1, \mathit{out}2, \mathit{out}3)) = \\
& \forall t.\ (\mathit{out1}\ t, \mathit{out2}\ t, \mathit{out3}\ t) = \\
& \qquad (\mathit{if}\ \mathit{IS\_NONE}(\mathit{inp}\ t)\ \mathit{then}\ \mathit{NONE} \\
& \qquad\ \mathit{else}\ (\mathit{FST}(\mathit{THE}(\mathit{inp}\ t))), \\
& \qquad\ \mathit{if}\ \mathit{IS\_NONE}(\mathit{inp}\ t)\ \mathit{then}\ \mathit{NONE} \\
& \qquad\ \mathit{else}\ (\mathit{SND}(\mathit{FST}(\mathit{THE}(\mathit{inp}\ t)))), \\
& \qquad\ \mathit{if}\ \mathit{IS\_NONE}(\mathit{inp}\ t)\ \mathit{then}\ \mathit{NONE} \\
& \qquad\ \mathit{else}\ (\mathit{SND}(\mathit{SND}(\mathit{THE}(\mathit{inp}\ t)))))
\end{aligned}
$$

Now we illustrate how we can compose the components shown above to build a more elaborate system. A *BLOCK* is a system that has a certain delay d, may break according to the fail function e, and computes f. A *BLOCK* is constructed as a sequential composition of *ERROR*, *COMB* and *DEL* (see Figure 1).

$$
\begin{aligned}
& \mathit{BLOCK}\ d\ e\ f\ (\mathit{inp}\ \mathit{out}) = \\
& \quad \forall t.\ \mathit{out}\ (t+d) = \exists \mathit{out1}\ \mathit{out2}.\ \mathit{ERROR}\ e\ (\mathit{inp}, \mathit{out}1) \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mathit{COMB}\ f\ (\mathit{out}1, \mathit{out}2) \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mathit{DEL}\ d\ (\mathit{out}2, \mathit{out})
\end{aligned}
$$

Internal signals *out1* and *out2* are existentially quantified. This quantification hides from the user of *BLOCK* the values of these signals, exposing only its external interface. Signals with the same name connect two components. For instance, *out1* is the output of *ERROR* and the input of *COMB*. This style of modelling block diagrams is fairly standard in the hardware verification domain [19].

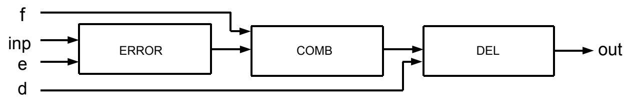


Fig. 1. The diagram of *BLOCK*

The theorem below shows that a $(BLOCK\ d\ e\ f\ (inp, out))$ implements a $(SYSTEM\ d\ e\ f\ (inp, out))$.

$$\vdash \forall d\ e\ f\ inp\ out.\ BLOCK\ d\ e\ f\ (inp, out) \Rightarrow SYSTEM\ d\ e\ f\ (inp, out)$$

The proof of this theorem in HOL4 is done by expanding all definitions, then eliminating the existential and universal quantifiers (see Camilleri *et al.* [5] for more details on how to do this in HOL4). After that, the proof is subdivided in cases and simplification tacticals are conveniently applied to finish the proof. All theorems in this work have been proved mechanically in HOL4. We omit further information on the proofs from now on.

In what follows we describe how we build fault tolerant patterns using the components described above.

3.3 Homogeneous Redundancy

This pattern is possibly the simplest existing fault tolerant pattern. The original channel is simply duplicated in order to improve reliability. The replicated channels operate in parallel, producing outputs at same time. The channels take input data from different sources. This pattern is called *homogeneous* because the replicas are exactly the same (same implementation, same technology, etc.). A monitor is used to implement a *switch-to-backup* policy in case an error occurs in the primary channel. This policy specifies that when the output produced by the primary channel is *NONE*, then the backup channel is used.

A homogeneous redundant system *HR* connects two systems *S1* and *S2* to a *BLOCK* with delay *dm*, fail function *em*, and functionality equals to the function *MONITOR* (described below). The input *inp* is split by *MUX* into two signals: *inpsys1* and *inpsys2*. These signals are the inputs of *S1* and *S2*, respectively. The output of both *S1* and *S2* is combined by *BUS* and sent to a monitor *BLOCK*, which decides which system is used.

$$\begin{array}{l}
HR\ dm\ em\ S1\ S2\ (inp : num \rightarrow (\alpha\ option \times \alpha\ option)\ option, out)\ = \\
\forall t.\ out\ (t+d) = \exists inpsys1\ inpsys2\ outsys1\ outsys2\ outbus. \\
\qquad\qquad\qquad MUX\ (inp, (inpsys1, inpsys2)) \wedge \\
\qquad\qquad\qquad S1\ (inpsys1, outsys1) \wedge \\
\qquad\qquad\qquad S2\ (inpsys2, outsys2) \wedge \\
\qquad\qquad\qquad BUS\ ((outsys1, outsys2), outbus) \wedge \\
\qquad\qquad\qquad BLOCK\ dm\ em\ MONITOR\ (outbus, out)
\end{array}$$

Note that the definition of *HR* does not force *S1* and *S2* to be duplicates of the same system. This will be done in the correctness theorem of *HR* in which we instantiate *S1* and *S2*.

The function *MONITOR* takes as input the current time and a pair of type $\alpha\ option \times \alpha\ option$ and decides which output to choose: the one from *S1* or the one from *S2*.

$$\begin{array}{l}
MONITOR\ t\ inp = if\ (IS\_SOME(FST(inp)))\ then \\
\qquad\qquad\qquad THE(FST(inp)) \\
\qquad\qquad else\ THE(SND(inp))
\end{array}$$

If the output from $S1$ is valid ($IS\_SOME(FST(inp))$) then $MONITOR$ returns the first element. Otherwise, it returns the second element. Note that $MONITOR$ is not a component. It instantiates the function f of a $BLOCK$.

The correctness theorem for HR is shown below. We assume that $I1$ and $I2$ are two implementations of a $SYSTEM$ that has delay d and computes the function f. $I1$ and $I2$ differ only in their fail functions $e1$ and $e2$, i.e. they do not necessarily synchronise on their random failures.[1] Given these two implementations of a $SYSTEM$ that computes f, the theorem states that if we plug them into the HR , the resulting system also implements a $SYSTEM$. Moreover, such $SYSTEM$ has a delay $d+dm$, a fail function $(E\ e1\ e2\ em\ d\ inp)$ and computes the function $(FHR\ f\ e1)$. The definitions of E and FHR are given below.

$$
\begin{array}{l}
\vdash \forall I1\ I2\ d\ e1\ e2\ f\ dm\ em\ inp\ out. \\
\quad (\forall inp\ out.\ I1(inp, out) \Rightarrow SYSTEM\ \ d\ e1\ f\ \ (inp, out)) \wedge \\
\quad (\forall inp\ out.\ I2(inp, out) \Rightarrow SYSTEM\ \ d\ e2\ f\ \ (inp, out)) \\
\qquad \Rightarrow (HR\ dm\ em\ I1\ I2\ (inp, out) \\
\qquad\qquad \Rightarrow SYSTEM\ (d + dm)\ (E\ e1\ e2\ em\ d\ inp)\ (FHR\ f\ e1)\ (inp, out))
\end{array}
$$

The fail function of an HR that comprises two channels $I1$ and $I2$ with fail functions $e1$ and $e2$, respectively, is given below.

$$
\begin{array}{l}
E\ e1\ e2\ em\ d\ inp\ t = \\
\qquad em(t + d) \vee \\
\qquad (e1(t) \wedge e2(t)) \vee \\
\qquad (e1(t)\ \wedge\ IS\_NONE(SND(THE(inp\ t)))) \vee \\
\qquad (e2(t)\ \wedge\ IS\_NONE(FST(THE(inp\ t)))) \vee \\
\qquad (IS\_NONE(SND(THE(inp\ t)))\ \wedge\ IS\_NONE(FST(THE(inp\ t))))
\end{array}
$$

The HR can fail under five different conditions: (i) the monitor block fails; or (ii) both duplicated channels $I1$ and $I2$ fail simultaneously; or (iii) $I1$ fails and the input for $S2$ is $NONE$; or (iv) $I2$ fails and the input for $I1$ is $NONE$; or (v) the input for both $I1$ and $I2$ are $NONE$ simultaneously.

The HR functionality is described by the function $(FHR\ f\ e1)$, where f is the functionality of both $I1$ and $I2$, $e1$ is the fail function of primary channel ($I1$) and t is the current time.

$$
\begin{array}{l}
FHR\ f\ e1\ t\ (inp : (\alpha\ option \times \alpha\ option)) = \\
\qquad if\ IS\_NONE(FST(inp)) \vee e1(t) \\
\qquad then\ (f\ t\ (THE(SND(inp)))) \\
\qquad else\ (f\ t\ (THE(FST(inp))))
\end{array}
$$

If the input for $I1$ is $NONE$ or $I1$ fails at time t, then HR behaves like $I2$ (it applies f to the input of $I2$). Otherwise, it behaves like $I1$ by applying f to the input of $I1$. As we need to check if there is a failure of $I1$ at time t, we take both the time and the fail function $e1$ as argument. Notice that in both cases, HR preserves the functionality of $I1$ and $I2$: the same function f is applied in both branches of the *if-then-else*.

[1] This assumption is in accordance to real duplicate systems: they do the same thing and they have the same fail *rates*, but they do not necessarily fail together.

3.4 Heterogeneous Redundancy

This pattern improves reliability by offering channels with dissimilar design or implementation, i.e. different design or implementation for systems that do the same thing. Dissimilarity channels are particularly useful to reduce the chances of replicating two systems with the same systematic failures (bugs).

The heterogeneous redundant system *HetR* specification is quite similar to *HR*, except that the inputs have different types as the replicated systems have different implementations. Their outputs, however, have the same types as they must compute the same thing. As in *HR*, the *HetR* connects two systems *S1* and *S2* to a *BLOCK* with delay *dm*, fail function *em* and that implements the function *MONITOR*. The monitor function is the same of *HR*.

$$\begin{aligned}
&\mathit{HetR}\ \mathit{dm}\ \mathit{em}\ S1\ S2\ (\mathit{inp} : \mathit{num} \rightarrow (\alpha\ \mathit{option} \times \beta\ \mathit{option})\ \mathit{option}, \mathit{out}) \ =\\
&\forall t.\ \mathit{out}\ (t+d) = \exists \mathit{inpsys1}\ \mathit{inpsys2}\ \mathit{outsys1}\ \mathit{outsys2}\ \mathit{outbus}\\
&\qquad \mathit{MUX}\ (\mathit{inp}, (\mathit{inpsys1}, \mathit{inpsys2})) \wedge\\
&\qquad \mathit{S1}\ (\mathit{inpsys1}, \mathit{outsys1}) \wedge\\
&\qquad \mathit{S2}\ (\mathit{inpsys2}, \mathit{outsys2}) \wedge\\
&\qquad \mathit{BUS}\ ((\mathit{outsys1}, \mathit{outsys2}), \mathit{outbus}) \wedge\\
&\qquad \mathit{BLOCK}\ \mathit{dm}\ \mathit{em}\ \mathit{MONITOR}\ (\mathit{outbus}, \mathit{out})
\end{aligned}$$

Note that the input is now of type $\mathit{num} \rightarrow (\alpha\ \mathit{option} \times \beta\ \mathit{option})\ \mathit{option}$. For *HR*, the input is of type $\mathit{num} \rightarrow (\alpha\ \mathit{option} \times \alpha\ \mathit{option})$. This is what captures heterogeneity in our model. As the difference between homogeneous and heterogeneous depends only on the type of input, perhaps *HetR* could cover both. We present both *HR* and *HetR* here as they are standard patterns for fault tolerance [8].

The correctness theorem is also very similar to the *HR*. If *I1* and *I2* are correct implementations of a *SYSTEM*, then an *HetR* system that contains *I1* and *I2* is also an implementation of a *SYSTEM*. Such *SYSTEM*, however, computes the function *FHetR* (shown below).

$$\begin{aligned}
&\vdash \forall \mathit{I1}\ \mathit{I2}\ d\ \mathit{e1}\ \mathit{e2}\ \mathit{f1}\ \mathit{f2}\ \mathit{dm}\ \mathit{em}\ \mathit{inp}\ \mathit{out}.\\
&\quad (\forall \mathit{inp}\ \mathit{out}.\mathit{I1}(\mathit{inp}, \mathit{out}) \Rightarrow \mathit{SYSTEM}\ d\ \mathit{e1}\ \mathit{f1}\ (\mathit{inp}, \mathit{out})) \wedge\\
&\quad (\forall \mathit{inp}\ \mathit{out}.\mathit{I2}(\mathit{inp}, \mathit{out}) \Rightarrow \mathit{SYSTEM}\ d\ \mathit{e2}\ \mathit{f2}\ (\mathit{inp}, \mathit{out}))\\
&\qquad \Rightarrow (\mathit{HetR}\ \mathit{dm}\ \mathit{em}\ \mathit{I1}\ \mathit{I2}\ (\mathit{inp}, \mathit{out})\\
&\qquad \Rightarrow \mathit{SYSTEM}\ (d + \mathit{dm})\ (E\ \mathit{e1}\ \mathit{e2}\ \mathit{em}\ d\ \mathit{inp})\ (\mathit{FHetR}\ \mathit{f1}\ \mathit{f2}\ \mathit{e1})\ (\mathit{inp}, \mathit{out}))
\end{aligned}$$

The function *FHetR* chooses between the functionalities provided by the replicas.

$$\begin{aligned}
&\mathit{FHetR}\ \mathit{f1}\ \mathit{f2}\ \mathit{e1}\ t\ (\mathit{inp} : (\alpha\ \mathit{option} \times \beta\ \mathit{option})) =\\
&\qquad \mathit{if}\ \mathit{IS\_NONE}(\mathit{FST}(\mathit{inp}))\ \vee\ \mathit{e1}(t)\\
&\qquad \mathit{then}\ (\mathit{f2}\ t\ \mathit{THE}(\mathit{SND}(\mathit{inp})))\\
&\qquad \mathit{else}\ (\mathit{f1}\ t\ \mathit{THE}(\mathit{FST}(\mathit{inp})))
\end{aligned}$$

If the input for *I1* is *NONE* or if *I1* breaks at time t, then the function *f2* of *I2* is used. Otherwise, *f1* is computed.

3.5 Triple Modular Redundancy

The triple modular redundancy is a variation of the homogeneous redundancy that consists of three identical channels that operate in parallel, and a voter that compares and averages the outputs of the channels. As in *HR*, only random faults can be addressed by this pattern, but it differs from *HR* by allowing the system to provide a valid output in the presence of up to two simultaneous random failures.

The voter plays a main role in this pattern by applying a voting policy that takes into account the majority of the valid outputs from the replicas [2]. We assume that the channel's outputs can diverge slightly, i.e. two *SOME* outputs do not need be exactly the same. This assumption reflects the fact that every channel receives its input from independent sources (typically, distinct sensors), and that these sources can produce slightly different values even using same technology. This happens specially when the sources output values of type real.

The *VOTER* shown below is a function that averages the valid outputs from the channels in order to minimise deviations. We assume that the type of the function computed by the channels is $num \rightarrow \alpha \rightarrow real$ as *real* arithmetic is needed in order to compute the average. For simplicity, the input *inp* is subdivided in three components: *in1*, *in2* and *in3*, these components refer to the input of three channel replicas. If all signals are valid, the *VOTER* outputs the arithmetic average of all them; in case of just one signal is invalid, this signal is disregarded and the average of other two is given as result. Finally, if just one signal is valid, this signal is the output itself.

```
$VOTER\ t\ inp =$
    $let\ in1 = FST(inp)\ in$
    $let\ in2 = FST(SND(inp))\ in$
    $let\ in3 = SND(SND(inp))\ in$
        $if\ IS\_SOME(in1) \wedge IS\_SOME(in2) \wedge IS\_SOME(in3)\ then$
          $(1/3) * (THE(in1) + THE(in2) + THE(in3))$
        $else\ if\ IS\_SOME(in1) \wedge IS\_SOME(in2)\ then$
          $(1/2) * (THE(in1) + THE(in2))$
        $else\ if\ IS\_SOME(in1) \wedge IS\_SOME(in3)\ then$
          $(1/2) * (THE(in1) + THE(in3))$
        $else\ if\ IS\_SOME(in2) \wedge IS\_SOME(in3)\ then$
          $(1/2) * (THE(in2) + THE(in3))$
        $else\ if\ IS\_SOME(in1)\ then\ THE(in1)$
        $else\ if\ IS\_SOME(in2)\ then\ THE(in2)$
        $else\ THE(in3)$
```

The *TMR* pattern connects three systems *S1*, *S2* and *S3* to a *BLOCK* with delay *dv*, fail function *ev* and that implements the function *VOTER*. Notice that *TBUS* and *TMUX* play the role of *BUS* and *MUX* used with *HR* and

HetR. As the channel replicas are homogeneous, the input is of type $num \rightarrow (\alpha\ option \times \alpha\ option \times \alpha\ option)\ option$.

$$\begin{array}{l} TMR\ dv\ ev\ S1\ S2\ S3\ (inp, out) = \\ \forall t.\ out\ (t + dv) = \exists inpsys1\ inpsys2\ inpsys3\ outsys1\ outsys2\ outsys3\ outbus. \\ \qquad TMUX\ (inp, (inpsys1, inpsys2, inpsys3)) \wedge \\ \qquad S1\ (inpsys1, outsys1) \wedge \\ \qquad S2\ (inpsys2, outsys2) \wedge \\ \qquad S3\ (inpsys3, outsys3) \wedge \\ \qquad TBUS\ (outsys1, outsys2, outsys3, outbus) \wedge \\ \qquad BLOCK\ dv\ ev\ VOTER\ (outbus, out) \end{array}$$

The correctness theorem states if *I1*, *I2* and *I3* are correct implementations of a *SYSTEM*, then a *TMR* system that contains *I1*, *I2* and *I3* is also an implementation of a *SYSTEM*. Such *SYSTEM*, has a delay $d+dv$, a fail function ($ETMR\ e1\ e2\ e3\ ev\ d\ inp$) and computes the function ($FTMR\ e1\ e2\ e3\ f$). The definitions of *ETMR* and *FTMR* are explained below.

$$\begin{array}{l} \vdash \forall I1\ I2\ I3\ dv\ ev\ d\ e1\, e2\ e3\ f\, inp\ out. \\ \quad (\forall inp\ out. I1(inp, out) \Rightarrow SYSTEM\ d\ e1\ f\ (inp, out)) \wedge \\ \quad (\forall inp\ out. I2(inp, out) \Rightarrow SYSTEM\ d\ e2\ f\ (inp, out)) \wedge \\ \quad (\forall inp\ out. I3(inp, out) \Rightarrow SYSTEM\ d\ e3\ f\ (inp, out)) \\ \qquad \Rightarrow (TMR\ dv\ ev\ I1\ I2\ I3\ (inp, out) \\ \qquad\qquad \Rightarrow SYSTEM\ (d + dv)\ (ETMR\ e1\ e2\ e3\ ev\ d\ inp) \\ \qquad\qquad\quad (FTMR\ e1\ e2\ e3\ f)\ (inp, out)) \end{array}$$

The *TMR* can fail in one of these cases: either i) all channels present a random failure simultaneously; or ii) the voter presents a random failure; or iii) there is a combination of *NONE* inputs and channel failures that involves all channel replicas; or iv) all inputs are *NONE*.

$$\begin{array}{l} ETMR\ e1\ e2\ e3\ ev\ dv\ inp\ t = \\ \quad let\ in1 = FST(THE(inp))\ in \\ \quad let\ in2 = FST(SND(inp))\ in \\ \quad let\ in3 = SND(SND(inp))\ in \\ \qquad (e1(t)\ \wedge\ e2(t)\ \wedge\ e3(t)) \vee \\ \qquad (ev(t + dv)) \vee \\ \qquad (IS\_NONE(in1)\ \wedge\ e2(t)\ \wedge\ e3(t)) \vee \\ \qquad (IS\_NONE(in2)\ \wedge\ e1(t)\ \wedge\ e3(t)) \vee \\ \qquad (IS\_NONE(in3)\ \wedge\ e1(t)\ \wedge\ e2(t)) \vee \\ \qquad (IS\_NONE(in1)\ \wedge\ IS\_NONE(in2)\ \wedge\ e3(t)) \vee \\ \qquad (IS\_NONE(in1)\ \wedge\ IS\_NONE(in3)\ \wedge\ e2(t)) \vee \\ \qquad (IS\_NONE(in2)\ \wedge\ IS\_NONE(in3)\ \wedge\ e1(t)) \vee \\ \qquad (IS\_NONE(in1)\ \wedge\ IS\_NONE(in2)\ \wedge\ IS\_NONE(in3)) \end{array}$$

The function *FMTR* takes 6 arguments: the fail function of each replica, the computing function of the replicas (which is the same function for all replicas), the input, and the current time. Based on the input and the fail functions, *FMTR* dismisses *NONE* outputs of each channel and outputs the average of the valid outputs. This function assumes the input *inp* has at least one valid signal. This assumption is always satisfied as *inp* comes from *TMUX*.

```
$FTMR\ e1\ e2\ e3\ f\ (t : num)\ inp\ =$
    $let\ in1 = FST(inp)\ in$
    $let\ in2 = FST(SND(inp))\ in$
    $let\ in3 = SND(SND(inp))\ in$
      $if\ IS\_SOME(in1)\ \wedge\ IS\_SOME(in2)\ \wedge\ IS\_SOME(in3)\ \wedge$
            $\neg e1(t)\ \wedge\ \neg e2(t)\ \wedge\ \neg e3(t)$
         $then\ (1/3) * (f\ t\ (THE\ in1) + f\ t\ (THE\ in2) + f\ t\ (THE\ in3))$
      $else\ if\ IS\_SOME(in1)\ \wedge\ IS\_SOME(in2)\ \wedge\ \neg e1(t)\ \wedge\ \neg e2(t)$
            $then\ (1/2) * (f\ t\ (THE\ in1) + f\ t\ (THE\ in2))$
      $else\ if\ IS\_SOME(in1)\ \wedge\ IS\_SOME(in3)\ \wedge\ \neg e1(t)\ \wedge\ \neg e3(t)$
            $then\ (1/2) * (f\ t\ (THE\ in1) + f\ t\ (THE\ in3))$
      $else\ if\ IS\_SOME(in2)\ \wedge\ IS\_SOME(in3)\ \wedge\ \neg e2(t)\ \wedge\ \neg e3(t)$
            $then\ (1/2) * (f\ t\ (THE\ in2) + f\ t\ (THE\ in3))$
      $else\ if\ IS\_SOME(in1)\ \wedge\ \neg e1(t)\ then\ (f\ t\ (THE\ in1))$
      $else\ if\ IS\_SOME(in2)\ \wedge\ \neg e2(t)\ then\ (f\ t\ (THE\ in2))$
      $else\ (f\ t\ (THE\ in3))$
```

Differently from *HR* and *HetR*, it is hard to see that triple modular redundancy preserves the essence of behaviour of the channels. The behaviour of *TMR* is the average of the behaviour of the parts. The only cases where the behaviour is preserved completely are those in which there is only one valid channel (see the last 3 *if* branches above). It is easy to define further variations on the voter by changing the definition of the *VOTER*.

4 Case Study

In this section we show a (very) simplified model of an aircraft *elevator* control system (ECS) translated from a Simulink diagram to a HOL4 function. Elevator surfaces control the aircraft's orientation by changing the up-and-down movement of the aircraft's nose (this movement is called *pitch*). For conciseness, we abstracted away several low level details of the ECS and assume the translation is correct. This assumption does not compromise our case study as our verification concerns the redundancy of the ECS instead of verifying the ECS itself. We show how the ECS function can be viewed as a block that is subsequently applied to the TMR pattern. The correctness theorem for the TMR is easily derived from the theorems shown in Section 3.

The elevator controller takes as input values from several sensors and outputs the command to the elevator.

$$\begin{array}{l}
elevator\ t\ (PitchRate, Flap, WOW, LongSideStick, PitchRate\_Voted) = \\
let\ out\_lpf = low\_pass\_filter(1, PitchRate)\ in \\
let\ out\_cpt = compensator(1/2, 1/4, out\_lpf)\ in \\
let\ out\_gfe = Gain(-150, out\_cpt)\ in \\
let\ out\_gfc = Gain(-67, out\_cpt)\ in \\
let\ out\_sth = SwitchThreshold(1/2, Flap, out\_gfe, out\_gfc)\ in \\
let\ out\_not = NOT(WOW)\ in \\
let\ out\_and = AND(out\_not, PitchRate\_Voted)\ in \\
\ldots \\
let\ out\_str = ElevSaturation(-25, 25, out\_sum) \\
in\ out\_str
\end{array}$$

The elevator controller takes as input the angular momentum of the airplane (*PitchRate*), the flap position (*Flap*), the weight on wheels (*WOW*), the longitudinal side-stick deflection (*LongSideStick*) and a signal that validates the pitch rate (*PitchRate_Voted*). The *low_pass_filter* filters low-frequency signals and reduces the amplitude of signals with frequencies higher than a certain cutoff frequency. The *compensator* increases the stability of the system response. The *Gain* amplifies the input by a certain amount given by the first argument. The *switchThreshold* allows one to choose between two inputs (*out_gfe* and *out_gfc*). The decision is made according to the value of a threshold (1/2) and the switch input (*Flap*). The functions *NOT* and *AND* represent boolean negation and boolean conjunction, respectively. The *ElevSaturation* imposes upper and lower bounds on a value. When the input value *PitchRate_Voted* is within the range −25 and 25, the output signal is equal to the input signal. Otherwise, it restricts the signal to the upper and lower bounds.

It is easy to prove that a *BLOCK* that implements the *elevator* is a *SYSTEM*.

$$\begin{array}{l}
\vdash \forall d\ e\ inp\ out.\ BLOCK\ \ d\ e\ elevator(inp, out) \\
\qquad\qquad\qquad\qquad \Rightarrow SYSTEM\ \ d\ e\ elevator(inp, out)
\end{array}$$

The proof of this theorem is a simple instantiation of the function f of the theorem shown at the end of Section 3.2.

In a similar way, we can instantiate the correctness theorem for the Triple Modular Redundancy in order to prove that the redundant system implements an *FTMR* that averages computation of the *elevator*.

$$\begin{array}{l}
\vdash \forall d\ e1\ \ e2\ \ e3\ \ ev\ \ dv\ \ inp\ \ out. \\
\qquad TMR\ dv\ ev\ (BLOCK\ \ d\ \ e1\ \ elevator)\ (BLOCK\ \ d\ \ e2\ \ elevator) \\
\qquad\qquad\qquad (BLOCK\ \ d\ \ e3\ \ elevator)\ (inp, out) \\
\qquad \Rightarrow\ SYSTEM\ (d + dv)\ (ETMR\ \ e1\ \ e2\ \ e3\ \ ev\ \ d\ \ inp) \\
\qquad\qquad\qquad\quad (FTMR\ \ e1\ \ e2\ \ e3\ \ elevator)\ (inp, out)
\end{array}$$

The proofs for these theorems are trivial. We only have to instantiate the function to be *elevator*. The proof effort was entirely on the proof of the theorems of Section 3. It is possible to make such proofs completely automatic, although we have not implemented it yet.

Note we do not need to instantiate the fail functions in the theorem above. It occurs due to the fact that the correctness theorem for *TMR* (and another patterns) universally quantify the fail function. This is an advantage of our model, since our theorem can be applied for systems with any fail function.

Thanks to the compositionality of our theorems, we can continue to apply more fault redundant patterns if we wish. For instance, we could replicate the entire *TMR* system and plug the replicas into an *HR*. As the *TMR* also implements a *SYSTEM*, the entire replication with *HR* could be easily proved to implement a *SYSTEM* too.

5 Related Work

This section presents related work on the formal verification of fault tolerant patterns. Pioneering work on the verification of fault tolerant patterns were done by Owre *et al.* [16], Butler *et al.* [4] and Sokolsky *et al.* [18] in the nineties. They verified a model for a fault tolerant architecture for distributed processors called the Reliable Computing Platform (RCP) and a redundancy management system (RMS) for a Space Launch Vehicle.

The formalisation of Owre *et al.* was done in EHDM [15]. Their verification proved that a replicated synchronous system using majority voting presents the same behaviour of a single system with no failures [16]. Their proof assumes that only transient failures happen and that damages to a data caused by one subsystem do not propagate to cause another working subsystem to failure.

A more ambitious and detailed verification was carried out by Butler *et al.* [4]. Their verification of the RCP is divided into 5 levels of specifications. The topmost level is the equivalent one to Owre's single ideal system. The levels below it describe the system as a synchronous replicated system, distributed replicated system (whose communication takes time), distributed asynchronous replicated system, and local executive (which takes into account details of the operating system memory management, task management, and inter-processor communication).

Sokolsky *et al.* verified a redundancy management system (RMS) for a Space Launch Vehicle [18]. The RMS verified is a modular architecture that separates two concerns: the redundancy management system is developed independently from the application. The verification was carried out in the process algebra ACSR [3]. The requirements were formalised and its properties were subsequently verified in the PARAGON toolset [17]. The verification was carried out by the PARAGON model checker. For every property to be verified, an observer process was created to run in parallel with the system. If an illegal behaviour was observed, the observer induced a deadlock in order to stop the checker.

More recently, Dajani-Brown *et al.* applied SCADE [1] (a commercial language and tool similar to Lustre [10]) to verify a triple modular redundancy with SCADE's model checker [7]. Errors are modelled in a similar way we do: an external signal indicates whether there must be a failure or not. Their model is more specific with respect to the timing of the failure. The model is configured to have failures not occurring simultaneously and with a well-defined duration. Similar to our model, noise is not formalised. Unlike ours, there is no failure on the voter itself.

The related works differ from ours mainly on the compositionality of the theorems, and on the model of errors. Owre *et al.* [16] and Butler *et al.* [4] assume that the original channel cannot fail and that, under certain conditions of the fail function, the replicated system behaves like a perfect original channel. We also capture the situation where a simultaneous failure occurs, differently from Dajani-Brown *et al.* [7].

6 Conclusion

We developed a model in HOL4 for fault tolerant patterns. Our model describes the original non-replicated channel as a system capable of: computing a certain function; subject to failure; and having an initialisation delay. The fault tolerant patterns also compute a certain function (which is essentially the computation done by the original system), are also subject to failure (according to a fail function that depends on the failure of its replicated channels), and have an initialisation delay.

The fail function for both the original channel and the fault tolerant pattern is the novel feature of our model. Previous works have assumed that the original channel and the fault tolerant patterns are perfect. We proved that the fault tolerant patterns preserve the essence of the original behaviour whenever the fail function evaluates to false. Moreover, the process of proving the correctness theorems also reveals us the fail logic: it is by proving the correctness theorem that we found out under which conditions the patterns can fail (i.e. the final definitions of the functions E and $ETMR$ were discovered during the process of proving). In addition, the compositionality of the theorems allows the user to compose fault tolerant patterns over and over in a correct way without any extra proof effort.

As future work, we intend to verify the correctness of fault tolerant patterns used in real projects in the industry. In particular, designs used by our partner Embraer, a Brazilian aircraft manufacturer. In addition, we plan to investigate to what extent our model applies to more general systems like block diagrams [12]. We also plan to verify variations of the patterns presented here.

Acknowledgement. We thank the National Institute of Science and Technology for Software Engineering (INES[2]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08. We also thank Embraer for the technical support and Robson Silva for review this paper.

[2] http://www.ines.org.br

References

1. The SCADE suite, http://www.esterel-technologies.com/products/scade-suite
2. Armoush, A.: Design Patterns for Safety-Critical Embedded Systems. Dissertation, Embedded Software Laboratory - RWTH Aachen University (June 2010)
3. Bremond-Gregoire, P., Lee, I., Gerber, R.: ACSR: An algebra of communicating shared resources with dense time and priorities. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 417–431. Springer, Heidelberg (1993)
4. Butler, R.W., Di Vito, B.L., Holloway, C.M.: Formal design and verification of a reliable computing platform for real-time control (phase 3 results). Technical memorandum 109140 (1994)
5. Camilleri, A., Gordon, M., Melham, T.: Hardware verification using higher-order logic. In: Borrione, D. (ed.) Proceedings of the IFIP WG 10.2 Working Conference on From HDL Descriptions to Guaranteed Correct Circuit Designs, pp. 43–67. North-Holland (1987)
6. Church, A.: A simple theory of types. Journal of Symbolic Logic 5, 56–68 (1940)
7. Dajani-Brown, S., Cofer, D.D., Bouali, A.: Formal Verification of an Avionics Sensor Voter Using SCADE. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 5–20. Springer, Heidelberg (2004)
8. Douglass, B.P.: Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Addison-Wesley Professional (2002)
9. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press (1993)
10. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE 79(9), 1305–1320 (1991)
11. The HOL4 System. SourceForge website, http://hol.sourceforge.net
12. International Electrotechnical Commission. IEC 61131-3 Ed. 1.0 en:1993: Programmable controllers — Part 3: Programming languages
13. Keith, L.: Advisory Circular - System Design and Analysis, 25.1309-1A (June 1988)
14. Koren, I., Krishna, C.M.: Fault Tolerant Systems. Morgan Kaufmann Publishers Inc., San Francisco (2007)
15. Melliar-Smith, P.M., Rushby, J.: The Enhanced HDM system for specification and verification. In: VerkShop III, Watsonville, CA, pp. 41–43 (1985)
16. Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification of fault-tolerant architectures: Prolegomena to the design of PVS. IEEE Transactions on Software Engineering 21(2), 107–125 (1995)
17. Sokolsky, O., Lee, I., Ben-Abdallah, H.: Specification and analysis of real-time systems with PARAGON (1999)
18. Sokolsky, O., Younis, M.F., Lee, I., Kwak, H.-H., Zhou, J.X.: Verification of the redundancy management system for space launch vehicle: A case study. In: IEEE Real Time Technology and Applications Symposium, pp. 220–229 (1998)
19. Melham, T.F.: Abstraction mechanisms for hardware verification. In: Birtwistle, G., Subrahmanyam, P.A. (eds.) VLSI Specification, Verification, and Synthesis, pp. 129–157. Kluwer Academic Publishers, Boston (1988)

A Formal Approach to Fixing Bugs*

Sara Kalvala and Richard Warburton

Department of Computer Science, University of Warwick, UK
{Sara.Kalvala,R.L.M.Warburton}@warwick.ac.uk

Abstract. Bugs within programs typically arise within well-known motifs, such as complex language features or misunderstood programming interfaces. Some software development tools often detect some of these situations, and some integrated development environments suggest automated fixes for some of the simple cases. However, it is usually difficult to hand-craft and integrate more complex bug-fixing into these environments. We present a language for specifying program transformations which is paired with a novel methodology for identifying and fixing bug patterns within Java source code. We propose a combination of source code and bytecode analyses: this allows for using the control flow in the bytecode to help identify the bugs while generating corrected source code. The specification language uses a combination of syntactic rewrite rules and dataflow analysis generated from temporal logic based conditions. We demonstrate the approach with a prototype implementation.

1 Introduction

Debugging existing programs while maintaining the *intent* of the programmer is an unavoidable but difficult task, which can take significant effort in the software development lifecycle. Some existing tools, such as FindBugs [8], can detect some of the commonly repeated bugs in particular programming languages, and some extensions to integrated development environments (IDEs), such as the UCDetector plugin [18], may attempt to suggest automated fixes for some of the simple cases. However, as far as we are aware, there is no general tool for specifying unusual or domain-specific bug detection mechanisms that also offers suggested fixes based on the specifications.

In this paper we propose a temporal-logic based language that offers a solution for this difficult problem of finding and fixing subtle bugs. Traditional application of abstract interpretation and static analysis is focused around checking a specified property of a specified program. In this work we seek to find bugs in large families of programs by facilitating the coding of common bug patterns and then detecting instances of those bug patterns. Each instance of a bug pattern is a potential bug and each pattern has one or more resolutions associated with it, that can be instantiated for a given potential bug. We use Java as our example platform, though our methodology is applicable to many imperative languages.

* This work was supported by the EPSRC under grant EP/DO32466/1 "Verification of the optimising phase of a compiler".

An important issue in writing static analysis systems is the representation over which the analysis is performed, notably whether at source code level, object code level or some intermediate representation. In order to bug-fix the programs themselves (rather than a low-level representation) it is necessary to perform the transformation at the *source code* level. There are many advantages, however, to performing analysis at a lower level: for example, it is easier to extract the control flow graph from a language whose control flow is represented by conditional goto statements, rather than loops. Therefore, many existing systems for detecting bugs perform analysis at the *bytecode* level, but then have difficulty incorporating fixes to source programs. We attempt to blend the best of both worlds with our approach to analysis: we perform syntactic analysis against the source code of the program, whilst performing semantic analysis on a bytecode representation. We use the standard debugging information from the Java Bytecode format in order to correlate the results from the source and Bytecode analyses.

The two characteristics of our work are therefore to support extensibility by allowing specification of new bug patterns, and correction of the original high-level programs. In this paper we show how to codify common bug patterns within a formally defined language based on temporal logic. We also simplify the construction of tools for static analysis of bug patterns, through model checking and rewriting.

In Section 2 we describe the kind of bugs which we consider and also the approach to software development for which our approach is particularly suited. We then describe, in Section 3, the language $\mathsf{TRANS}_{\mathsf{fix}}$ which can be used for both identifying bugs and implementing the transformations which correct the bugs. The prototype implementation FixBugs which applies bug fixes written in $\mathsf{TRANS}_{\mathsf{fix}}$ to Java programs is described in Section 4.

2 Methodology and Application

2.1 Example Bug Patterns and Categories

We use as a starting point the classification of common Java bugs due to Hovemeyer and Pugh [8], used in the description of the FindBugs tool which detects most of them. Many of the bugs identified by Hovemeyer and Pugh are simple and their identification requires merely a syntactic pattern matching system. Some of them, however, don't have obvious fixes. We especially consider some concurrency bugs, since they require more than simple syntactic pattern matching to be identified yet are amenable to temporal analysis.

Because of space limitations, in this paper we consider only three examples:

Method does not release lock on all paths. This bug arises in a situation where a method acquires a lock, but there exists a path through the method where the lock isn't released. The `java.util.concurrent` lock, as specified in JSR-166, is considered by the authors of FindBugs. Fig. 1 illustrates the standard solution to this bug.

```
Lock l = ...;
l.lock();
try {
    // do something
} finally { l.unlock(); }
```

Fig. 1. Pattern for correct locking

```
BufferedReader in = null;
    try {
        in = new BufferedReader(
            new FileReader(``foo''));
        String s;
        while((s=in.readLine()) != null) {
            System.out.println(s);        }
// (1) close mistakenly placed
        in.close();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
// (2) the close should be placed with guard by a null check
        if(in != null) {
            try { in.close();
            } catch (IOException e) {
                e.printStackTrace();  } } }
```

Fig. 2. Possibly Unclosed File Handle

Method may fail to close stream. This bug occurs when a method creates an IO stream object but does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. Good programming discipline requires the use of a `finally` block to ensure that streams are closed. Fig. 2 shows an example of (1) where not to place a close and (2) where to place it correctly.

Failed database transactions may not be rolled back. The JDBC library for database connections models the beginning, committing and ending of transactions through explicit calls to methods. A common bug pattern is a failure to check whether a transaction needs to be rolled back if its commit fails. The correct pattern is illustrated in Fig. 3. Another common problem is the failure to ensure that all paths end in either a commit or a rollback.

2.2 Placing Debugging within Software Development

In general, a good approach to process the fixing of bugs is to not entirely automate the application of transformations to the users' programs, since fixes may not always be semantics preserving. But if an automated tool is not designed

```
try {
    conn.setAutoCommit(false):
    ....
    conn.commit();
} catch(java.sql.SQLException e) {
    if(conn != null) {
        try {
            conn.rollback();
        } catch (java.sql.SQLException e) {
            e.printStackTrace();        }   }   }
```

Fig. 3. JDBC Commit and Rollback Pattern

to consider the specification of the program, there is the risk of introducing new bugs into a currently working system. Bug patterns usually identify scenarios that are *likely* to be buggy, rather than being *guaranteed* to be so. In this context, the conservative approach, which we adopt, is to not alter the program, but simply suggest bug fixes to the user.

Our implementation, described in Section 4 and which we call FixBugs, uses the Eclipse toolkit's intermediate representation to perform program transformation. This enables the production of source code that is formatted according to users' preferred style guidelines and integrates into the context in which programs are being developed.

While we have incorporated a few common bugs into FixBugs, the aim is to provide a *framework* in which more bugs can be accounted for. The designing of new transformations is easier than in traditional static analysis systems since the programmer does not have to implement new detailed analysis and transformation steps. Since the program transformations themselves are merely syntactic substitutions, it should be relatively natural for any experienced programmer to tailor the system to common bugs in their application area. The temporal logic side conditions may be considered a difficult notation to grasp, but we believe it is a simpler and more intuitive way of formulating dataflow analysis, than hand writing the code directly.

The FixBugs approach is not intended to subsume traditional debugging techniques such as testing, or traditional formal analysis techniques such as static analysis and model checking. Its integration into existing tools and techniques should complement their usage, allowing automated FixBugs sweeps of the code to be made in order to offer potential improvements to the code base. Bugs can be found as early as possible through these automated tools, rather than being identified later through failing test cases, often at a much higher cost. The inclusion within the development cycle of phases dedicated to improving code quality, such as the refactoring phases promoted by some agile methodologies, provides bug fixing program transformations with a suitable hook on which to integrate themselves into current practice.

3 A Language for Detecting and Fixing Bugs

3.1 Basis: The TRANS Language

In previous work concerned with the application of formally specified optimizations on Bytecode programs [20], we developed and extended Lacey's TRANS language [11,9]. In TRANS, compiler optimisations are represented through two components: a rewrite rule and a side condition which indicates the situations in which the rewrite can be applied safely.

Side conditions are expressed in an extension of CTL [4], a path-based temporal logic which can capture many properties while still being efficient to model-check. Temporal logics traditionally describe properties of a system relative to a point in time, but in TRANS the points of interest are nodes (or program points) in a *control flow graph* (or CFG) representing a program. The variant of CTL used includes past temporal operators ($\overleftarrow{E}$ and $\overleftarrow{A}$), the final operators EF and AF, and the henceforth operators EG and AG. The next state operators are extended with information on the kind of edge they operate over: for example, EX_{seq} and AX_{branch} stand for "there exists a next state via a *seq* edge" and "for all next states reached via a *branch* edge" respectively.

A logical judgement of the form: ϕ @ n states that the formula ϕ is *satisfied* at node n of the control flow graph. Two types of these basic predicates can be used to obtain information about a node in the control flow graph. The formula $node(x)$ holds at a node n in a valuation that maps n to x. The formula $stmt(s)$ holds at a node n where the valuation makes the pattern s match the statement at node n. As well as judgements about states, the language can make "global" judgements. For example, the formula ϕ @ $n \ \wedge \ conlit(c)$ states that ϕ holds at n and c is a constant literal throughout the program.

User defined predicates can be incorporated via a simple macro system. These can be used in the same way as core language predicates, and are defined by an equality between a named binding and the temporal logic side condition that the predicate should be 'expanded' into.

3.2 From TRANS to TRANS$_{\mathsf{fix}}$

We describe a variant of the TRANS language, called TRANS$_{\mathsf{fix}}$, suitable for specifying the transformation of Java source code with the aim of correcting bugs that may appear within programs. In contrast to TRANS, where the goal is to produce optimized *low-level* code, TRANS$_{\mathsf{fix}}$ is used to produce *source code*, since the goal of debugging is usually to maintain reusable and readable source code, for the developers of the software to continue working on. So rather than operating on the low-level code which is used as input for the temporal logic side conditions, rewrite rules must operate on the *source program* itself.

TRANS$_{\mathsf{fix}}$ specifications consist of actions and side conditions: if the side condition holds then the action is applied. Many actions consist of replacing statements with other statements, although they can also include adding new methods to classes. Actions are applied if side conditions hold.

A BNF for the $\mathsf{TRANS_{fix}}$ pattern matching language is provided in Fig. 4. Interesting aspects of $\mathsf{TRANS_{fix}}$ are its use of metavariables, the new actions and strategies, and the type system. The core syntax of the rewrite rules is based on standard programming constructs (assignment statements, while statements, if statements, etc) which we assume are well understood. The syntax is expanded with constructs to support meta-variables, representing either syntactic fragments of the program or nodes of the CFG.

The language for transformations contains a Java statement grammar, extended with metavariables that can bind to different program structures. For example, the pattern for matching an assignment of a variable by an addition expression, that is later followed by re-assignment to that variable, is shown in Fig. 5(a). The code snippet shown in Fig. 5(b) matches that pattern, via the bindings shown in Fig. 5(c).

$\mathsf{TRANS_{fix}}$ also contains a wildcard operator "...." that matches against any statement or (possibly empty) sequence of statements. Since a wildcard statement is a normal pattern matching statement, it can also be bound using a label, allowing the matching of arbitrary blocks of code in strategic locations. In order to facilitate the writing of specifications that are intuitive to programmers, we also allow wildcards to be used in the reconstruction of statements. This is syntactic sugar for binding the wildcard statements to metavariables using labels, and then substituting in metavariable references within the reconstruction pattern. Wildcard substitutions are indexed: the nth wildcard block in pattern matching is substituted into the nth wildcard position in the reconstruction pattern.

A consequence of the desire to produce source code is the necessity of incorporating *scoping*; while scoping doesn't exist within methods at a bytecode level, it is a necessary part of the transformation language of $\mathsf{TRANS_{fix}}$. Support of scoping allows us to match programming language constructs such as `try` and `catch` blocks.

Java types are also supported in the pattern matching. The pattern `:: m` binds any type to the metavariable `m`. One can explicitly refer to primitive types (such as `int`) or object types (such as `java.util.Vector`). One can also match arrays. The two `new` calls within the expressions grammar specifically allow pattern matching array initialisers.

3.3 Actions

Simple rewriting merely replaces code fragments with new code, but many transformations must actually change the structure of the `class` or apply rewrites at multiple places. These structural changes are supported by additional *actions*.

The `ADD_METHOD` action takes the return type of the method, its name, arguments and a statement to act as the body. This code is then added to a class, specified through a *metavar*. This is our primary method of transforming classes.

The `COMPOSE` action performs sequential composition on the two actions passed as arguments and forms a new atomic action. (This is not to be confused with the `THEN` *transformation* (see below) for composing two transformations.)

type ::= :: *metavar* | *primitive-type* | *object-type* | *type* `[]`

expr-pattern ::= *metavar* (*expression, expression* ...)?
| *expression op expression* | *unop expression*
| (*type*) *expression* | *expression* `instanceof` *type*
| `new` *type expression* | `new` *type* `[]`

statement ::= *metavar*: *statement* | ' *metavar* ' | . . . . | { *statement** }
| *type metavar* = *expression*
| `if` *expression statement statement*
| `while` *expression statement*
| `try` *expression* `catch` *statement* `finally` *statement*
| *expression* ; | `return` *expression* ; | `throw` *expression* ;
| `synchronized` (*expression*) { *statement* }
| `for` (*expression*, expression, expression**) { *statement* }
| `switch` (*expression*) { *statement** }
| `case` *expression*: *statement* ; | `default` ;
| `assert` *expression* ; | `continue` *metavar* ; | `break` *metavar*? ;
| `this` (*expression, expression* ...);
| `super` (*expression, expression* ...); | ;

node-condition ::= μ *condition-var*. *node-condition*
| ν *condition-var*. *node-condition*
| *node-condition* $\vee$ *node-condition*
| *node-condition* $\wedge$ *node-condition*
| $\neg$ *node-condition*
| $\exists$ *metavar* . *node-condition* | node(*metavar*)
| $[EX \mid AX \mid \overleftarrow{EX} \mid \overleftarrow{AX}]_{[metavar]}$ (*node-condition*)
| $[EF \mid AF \mid EG \mid AG]$ *node-condition*
| $[E \mid A \mid \overleftarrow{E} \mid \overleftarrow{A}]$ (*node-condition* U *node-condition*)

side-condition ::= *side-condition* $\vee$ *side-condition*
| *side-condition* $\wedge$ *side-condition*
| $\neg$ *side-condition*
| *node-condition* @ *metavar*
| *pred* ($metavar_1, \ldots, metavar_n$)

action ::= REPLACE *statement** WITH *statement**
| COMPOSE *action* WITH *action*
| CHOOSE *action* OR *action*
| ADD_METHOD *type metavar*(
    type metavar, ...) *statement* TO *metavar*

transform ::= *action* WHERE *side-condition*
| MATCH *side-condition* IN *transform*
| APPLY_ALL *transform*
| *transform* $\Box$ *transform*
| *transform* THEN *transform*

Fig. 4. BNF for TRANS$_{\mathsf{fix}}$

```
n: int x = l + r;
 ....
m: x = e;
```

(a) the pattern

```
int z = y + 5;
System.out.println(x);
z = z + 1;
```

(b) matching code

x	z
l	y
r	5
e	z + 1

(c) bindings

Fig. 5. TRANS$_{\text{fix}}$ Pattern Matching

Combining uses of actions has many applications, for example one could rewrite a block of code into a method, and replace it with a call to this method, by using a `REPLACE` composed with an `ADD_METHOD`.

A non-deterministic choice action, called `CHOOSE ... OR`, is used when the same analysis might suggest more than one possible fix. This fits in with the methodology of debugging we propose since the user must confirm the application of a transformation, and can be given several choices.

Transformationss are operators for combining different actions. The MATCH ϕ IN T transformation restricts the domain of information in the transformation T by the condition ϕ. The T_1 THEN T_2 transformation applies the sequential composition of T_1 and T_2. When actions are applied normally, ambiguity with respect to what node actions and rewrites are applied to are automatically resolved. In other words, if there are several bindings that have the same value for a node attribute that is being used in a rewrite rule then only one of them is non-deterministically selected. The APPLY_ALL T transformation uses all of the valuations within transformation T, without this restriction.

3.4 Type System

TRANS$_{\text{fix}}$ has a simple type system to ensure that programs transformed by a TRANS$_{\text{fix}}$ specification are syntactically valid Java programs. For example, anything nested at an expression level is an expression. In order to differentiate types of meta-variables being used in transformations from the types of Java variables, we refer to the former types as *kinds*. There are three types of *kinds*: `Type Kind` for metavariables that bind to Java types, `Expression Kind` for metavariables used for Java expressions, and `Statement Kind` for statements and blocks. The kind system guarantees two important properties:

1. that no metavariable may bind to, or substitute into a position that requires more than one Kind, and
2. that no metavariable may be used in a substitution, if it is not bound beforehand.

A relatively simple algorithm is used to check these properties. The syntactic replacement rules and side conditions are examined, keeping note of what context

```
REPLACE
        l: m.lock()
        ....
        u: m.unlock()
WITH
    try {
        m.lock()
        ....
    } finally { m.unlock() }
WHERE
        EF (node(u)) ∧ ¬AF (node(u)) @ l
```

Fig. 6. Transformation to ensure lock released on all paths

a metavariable is used in. If a metavariable is used in a context that implies it would need to be of more than one Kind, then kind-checking fails. If there are metavariables referred to in the substitution part of a replacement that aren't bound by either the pattern matching or the side condition then also the kind-checking fails.

3.5 Specification Examples

We re-visit the common bugs explained in Subsection 2.1 and show how typical fixes can be expressed in $\mathsf{TRANS}_{\mathsf{fix}}$.

Method does not release lock on all paths. The full specification is shown in Fig. 6. Position `l` within the program matches the point at which the lock is locked, and `u` at the position where it is unlocked. The side condition holds where you can sometimes unlock if you have locked, but not on every path. The replacement rule moves the unlock statement within a finally clause, ensuring that the lock gets executed on all paths through the method.

Method may fail to close stream. Fig. 7 gives a specification for rearranging the closing mechanism for file handles. It matches the type of the stream object into the metavariable `streamtype` and ensures this is a stream in the side condition. The other component of the side condition ensures that the close method throws an exception. Wildcard matching is used to keep the body of the `try` block in place, while moving the `close` call at the end of the method within a `finally` block—therefore ensuring that there is a path where the `close` method throws an exception.

Failed database transactions may not be rolled back. A specification for ensuring that transactions are surrounded by the correct catch pattern for `SQLException` instances is shown in Fig. 8. The pattern matching of a call to the `setAutoCommit` method matches the beginning of the transaction. The wildcard binds to anything between that and the `commit` call, *i.e.* a whole transaction. This block of code is then replaced with another block,

```
REPLACE
    ::streamtype stream = null;
    try {
        ....
        thro: stream.close();
    } catch (ex e) {
        c: ....   }
WITH
    ::streamtype stream = null;
    try {
        ....
    } catch (ex e) {
        ....
    } finally {
        if(stream != null) {
            try {
                stream.close();
            } catch('IOException' e) {
                e.printStackTrace(); }  }  }
WHERE
            subtype(streamtype,'java.io.OutputStream') ∧
                        EF (node(c)) @ thro
```

Fig. 7. Closing File Handles

surrounded by a `catch` statement. The `catch` statement rolls back the transaction in case of a database failure. The side condition checks to ensure that the `commit` call can never be followed by a `rollback`. It also ensures that `conn` is of the correct type.

4 Prototype Implementation

The approach proposed in this paper has been prototyped in the implementation we call FixBugs. This implementation takes a Java program in both source and Bytecode form and applies transformations to the source, outputting a series of programs representing possible bug-fixed variants of the program.

As shown in Fig. 9, the FixBugs system comprises several components. The Pattern Matcher produces bindings to metavariables from source code and a pattern, the Model Checker produces bindings to metavariables that satisfy the side condition formulae, and the Generator alters the program itself, given bound metavariables, according to the actions.

The Java programs source code is parsed using the Eclipse [5] project's Java developer tools, which provide a standardised intermediate representation for the programs. This representation is also manipulated by the Generator to produce bug-fixed programs in concrete syntax. The Model Checker relies on the ASM bytecode library [2] in order to generate the control flow graph of the program. ASM allows the manipulation of Bytecode at a programmer-friendly level of abstraction.

```
REPLACE
          conn.setAutoCommit(false):
          ....
commit:   conn.commit();
WITH
     try {
          conn.setAutoCommit(false):
          ....
          conn.commit();
     } catch(java.sql.SQLException e) {
          if(conn != null) {
               try {
                    conn.rollback();
               } catch (java.sql.SQLException e) {
                    e.printStackTrace();   }   }   }
WHERE
  type(conn,'java.sql.Connection') ∧¬EF(stmt(conn.rollback();))@ commit
```

Fig. 8. Correction for JDBC Commit and Rollback Pattern

4.1 Silhouettes

One line of Java source code is typically compiled into several lines of Java Bytecode. Consequently there is a mismatch in the level of detail when using the debugging information to bridge the analysis results of these two representational levels. We unify these levels within FixBugs through the concept of a *silhouette*. The silhouette of a statement of source code is the corresponding set of commands of its bytecode. The control flow graph silhouette of a source code line is the subgraph within the control flow graph that corresponds to that source code line. Every edge within the control flow graph of the program's source code has a corresponding edge within the bytecode control flow graph (but the inverse relation does not hold).

Silhouettes consequently partition the Bytecode control flow graph into several overlapping subgraphs. The edges between these subgraphs fall into two categories. An edge *(from,to)* is *inbound* with respect to some silhouette S if the *to* node, but not the *from* node, is a member of S, it is *outbound* if the *from* node is a member of S, but not *to*. If both *from* and *to* are within S we say that the edge is *contained* within S. The relation between source code and bytecode CFGs is illustrated in Fig. 10.

We can obtain the Java control flow graph from the Bytecode representation very simply with the following steps:

1. extract Bytecode control flow graph (G) using ASM.
2. compute line numbering function (L) using ASM.
3. coalesce (G) to form (G').

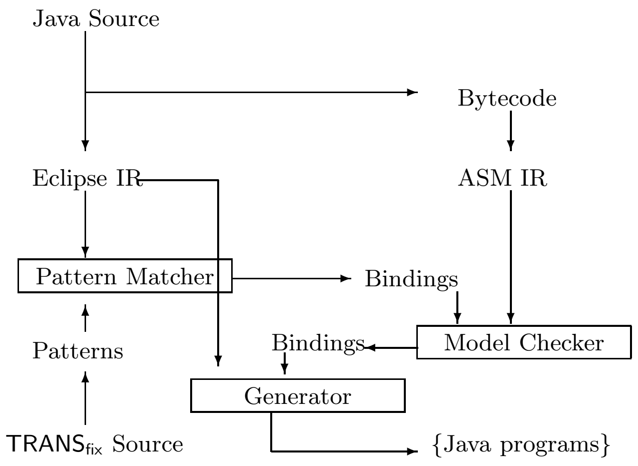


Fig. 9. FixBugs Architecture

Within FixBugs we represent the successor function of G as a map from integers onto sets of integers, and L as an array of integers. In order to calculate G' we therefore replace every edge *(from,to)* in G with an edge *(L(from),L(to))*. This ensures all inbound and outbound edges are replaced accordingly. We then remove all edges whose *from* and *to* nodes are identical, since they represent contained edges that don't exist within the source code control flow graph G'.

The use of the ASM Bytecode analysis library makes it easier to extract and coalesce the control flow graph than by writing a custom source code analysis. It also allows us to integrate other information more easily extracted at a Bytecode level, and then relabel it onto the Java control flow graph accordingly.

4.2 Implementation Details

FixBugs is coded primarily in Scala [15], chosen because of its support for a functional style of programming, combined with the plentiful libraries that are available on the Java platform. Specification files are parsed using the parser combinators in Scala's standard library. Disjoint union datatypes, modelled using case classes, provide an intermediate representation for TRANS$_{fix}$ specifications. Scala's pattern matching can then be used in order to bind TRANS$_{fix}$ metavariables to elements of Java source code, represented using Eclipse's Intermediate Representation. This development approach is described in Fig. 11.

Being a prototype, the current implementation doesn't provide support for all the features of the TRANS$_{fix}$ language, such as strategies and class-level actions. The gist of the approach, however, should map directly to these concepts, albeit with some programming effort.

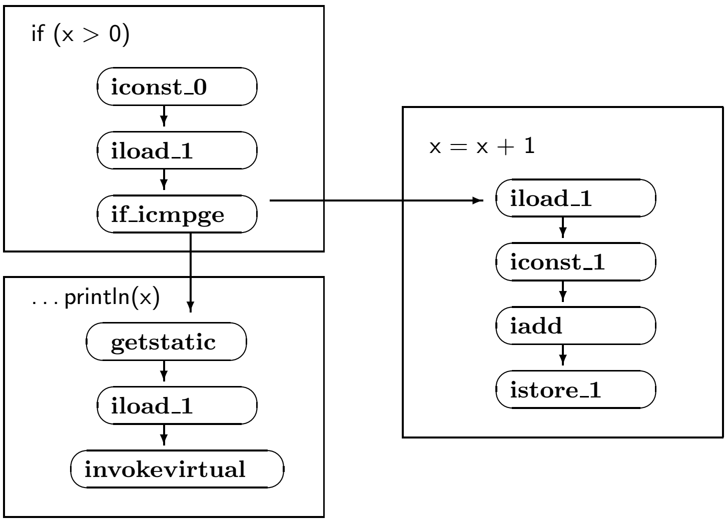


Fig. 10. Relating Java statements with the control flow graph

5 Analysis

We have introduced an approach that allows one to specify static analyses that can be applied to programs, and transformations that can be used to debug the programs. We describe a tool that allows the automated application of these transformations to programs and how its use can be integrated into existing development methodologies. Our implementation uses a novel technique for combining source code and object code analysis through *silhouettes*—a technique for unifying information annotated onto a control flow graph. This exploits the same underlying model as the $\mathsf{TRANS}_{\mathsf{fix}}$ specification language for transformations.

While we are satisfied with the performance of this prototype implementation in practice (applying the bug fixing transformations usually takes in the order of seconds) we have yet to complete an analysis of its computational complexity. CTL is polynomial time checkable in the size of the system times the length of the formula [3]. These correspond to the number of statements in the program being transformed, and the side condition of the transformational specification. Our pattern matching and reconstruction implementations are both linear in the size of the pattern plus the size of the method.

Before releasing the software to potential users, we intend to complete the following tasks:

1. Improve performance by making use of some existing symbolic model checker or boolean satisfiability solver.
2. Complete the implementation of language features, for example schematic variables and strategies, and extend to consider inter-procedural analysis.
3. Integrate into IDEs, in order to be able to use the tool effectively.

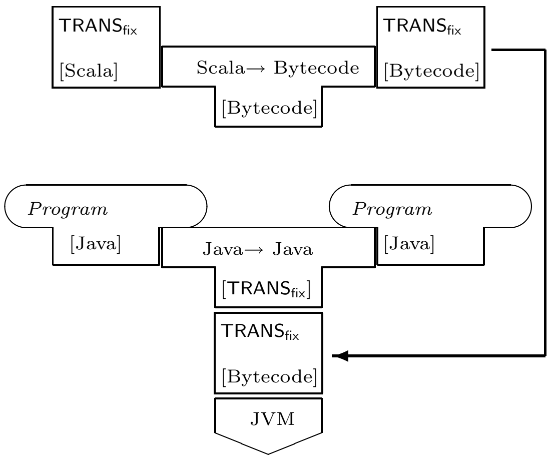


Fig. 11. Transformational Diagram for FixBugs

5.1 Related Work

In Section 2 we mentioned FindBugs, a system for detecting bugs within Java programs [8]. Bug patterns are defined as common constructs within programs that often causes errors. FindBugs detects these patterns through static analysis, but does not attempt to fix them. Its bug detection mechanisms are hand written in Java. UCDetector [18] is a plugin for the Eclipse IDE that finds unecessary code within a project. Its detection mechanism is a custom dead code static analysis. It can also detect when the visibility of a method can be restricted, for example from `public` to `private`. It can automatically fix the dead code issues that it detects, but only performs limited program analysis.

The use of predicates to identify program repair points is the basis of the work of Samanta *et al* [16]. Their approach relies on the use of standard pre- and post-conditions for a Boolean program and using propagation based on Hoare logic. This approach allows them to repair concurrent and recursive programs, and to reason about correctness. However, they haven't yet illustrated the approach on a full programming language, and do not show how language designers could extend the approach themselves by specifying new bug patterns.

Dataflow analysis has long been employed within the compiler optimisation community to iteratively compute the nodes within a program at which optimisations can be soundly applied [1,14]. Schmidt and Steffen explain the strong link between dataflow analysis and model checking, and show how equations for dataflow analyses can be expressed in modal μ-calculus [17]. Steffen also shows how dataflow analysis algorithms can be generated from modal logics [19]. Rewrite rules with temporal conditions have also been used in the Cobalt system [12] which focuses on automated provability and also provides executable specifications, achieved through temporal conditions common to many dataflow analysis approaches. The specific nature of Cobalt's temporal conditions is limited compared to the flexibility provided in TRANS$_{\text{fix}}$ from supporting CTL side conditions, even if this may require more expensive model checking. Rhodium is another domain specific language for developing compiler optimisations [13]. Rhodium consists of local rules that manipulate dataflow facts. This is a significant departure in approach from TRANS, since it uses more traditional, dataflow analysis based specifications rather than temporal side conditions. The Temporal Transformation Logic (TTL) [10] also uses CTL, but emphasizes verification of the soundness of the transformations themselves, *i.e.* that they are semantics preserving.

5.2 Correctness Issues

Unlike compiler optimisations, transformations applied to fix bugs are not semantics preserving. The very aim of the transformation is to alter the program semantics in order to remove a bug. Consequently one is assuming that the program itself is incorrect according to some specification, but can be corrected to match this specification. It is possible that the program itself might be correct, and accordingly the transformations should not be applied automatically. Additionally the bug finding patterns that we focus on correspond to behaviours that are generally considered bugs within a program, for example deadlocks.

We plan to extend our methodology to identify transformations that can be applied soundly, rather than simply leaving the choice of whether to apply these transformations to the user of the tool. The required soundness properties could be annotated onto the program. For example our specification for ensuring that locks are released on all paths is sound *iff* the user of the system wishes a lock to be in a released state as a post-condition of the method. Information of this nature can already be added to Java programs using the existing annotations framework, recently extended by [6]. There are already tools for invariant detection in partially annotated Java programs, [7] infers properties about nullness of variables. Another element of such an extension would be the ability to automatically infer the soundness of transformations with respect to given pre and post conditions.

However, we recall that bug-repairing transformations often have to change the semantics of a program, and the goal of a formal tool should be seen primarily to facilitate the development of correct programs, rather than be constrained by existing specifications. This is the approach supported by the FixBugs tool.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Pearson Education, London (2007)
2. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a code manipulation tool to implement adaptable systems. In: Adaptable and Extensible Component Systems (2002)
3. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8, 244–263 (1996)

4. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs, Workshop, pp. 52–71. Springer, London (1982)
5. Eclipse Foundation. Eclipse website (2009), http://www.eclipse.org
6. Ernst, M.D.: Type Annotations Specification (JSR 308), http://types.cs.washington.edu/jsr308/ (October 5, 2009)
7. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1-3), 35–45 (2007)
8. Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM SIGPLAN Notices 39(12), 92–106 (2004)
9. Kalvala, S., Warburton, R., Lacey, D.: Program transformations using temporal logic side conditions. ACM Transactions on Programming Languages and Systems (TOPLAS) 31(4) (2009)
10. Kanade, A., Sanyal, A., Khedker, U.: A PVS based framework for validating compiler optimizations. In: SEFM 2006: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods. IEEE Computer Society, Washington, DC (2006)
11. Lacey, D.: Program Transformation using Temporal Logic Specifications. PhD thesis, Oxford University Computing Laboratory (2003)
12. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. ACM Press, New York (2003)
13. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: POPL 2005: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 364–377. ACM Press, New York (2005)
14. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
15. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, 2nd edn. Artima Press (2010)
16. Samanta, R., Deshmukh, J.V., Emerson, E.A.: Automatic generation of local repairs for boolean programs. In: FMCAD (2008)
17. Schmidt, D.A., Steffen, B.: Data-flow analysis as model checking of abstract interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503. Springer, Heidelberg (1998)
18. Spieler, J.: UCDetector: the Unnecessary Code Detector website (2007), http://www.ucdetector.org
19. Steffen, B.: Generating data flow analysis algorithms from modal specifications. Science of Computer Programming 21, 115–139 (1993)
20. Warburton, R., Kalvala, S.: From specification to optimisation: An architecture for optimisation of Java bytecode. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 17–31. Springer, Heidelberg (2009)

A Formal Treatment of Agents, Goals and Operations Using Alternating-Time Temporal Logic

Christophe Chareton, Julien Brunel, and David Chemouil

Onera – The French Aerospace Lab,
F-31055 Toulouse, France
`firstname.lastname@onera.fr`

Abstract. The aim of this paper is to provide a formal framework for Requirements Engineering modelling languages featuring agents, behavioural goals and operations as main concepts. To do so, we define KHI, a core modelling language, as well as its formal semantics in terms of a fragment of the multi-agent temporal logic ATL*, called ATL_{KHI}. Agents in the sense of concrete and provided entities, called actors, are defined by their capabilities. They also pursue behavioural goals that are realised by operations, which are themselves gathered into abstract, required, agents, that we call roles. Then a notion of assignment, between (coalitions of) actors and roles is defined. Verifying the correctness of a given assignment then reduces to the validity of an ATL_{KHI} formula that confronts the capabilities of (coalitions of) actors with the operations in roles played by the said actors. The approach is illustrated through a toy example featuring an online shopping marketplace.

1 Introduction

Requirements Engineering (RE) is that part of software or systems engineering concerned with describing the problem domain and determining requirements for a system to be developed [10]. An important part of the RE discipline is concerned with isolating concepts for RE modelling. In this setting, the aim of this paper is to contribute to the formalisation of RE modelling languages featuring agents, behavioural goals and operations as main concepts.

Goals are a central concept of RE modelling languages. They are characterised as prescriptive statements over a system under study. Most of the time, goals are expressed in natural language. However, some approaches propose a framework to describe classes of goals formally. In particular, KAOS defines *behavioural goals* [10, 12, 13] in terms of a variant of Linear Temporal Logic (LTL) [15].

A goal may be refined into a combination of several sub-goals and, possibly, domain properties which are descriptive statements about the domain environment. Alternative refinements for a given goal can also be expressed.

In KAOS, the process of refining high-level goals into more precise ones ends with the production of two different kinds of goals: requirements and expectations. Requirements (resp. expectations) are under responsibility of agents in the software (resp. the environment). Requirements corresponding to behavioural goals are then realised by *operations*. In other approaches, the concepts of commitments [6] or tasks [18] play an analogous role.

In KAOS, an operation is defined by descriptive *domain pre-* and *post-conditions* that characterise its effects. Furthermore, the execution of an operation is constrained by prescriptive *required pre-*, *post-* and *trigger conditions* (trigger conditions are *sufficient* conditions for the execution of an operation, while preconditions are *necessary* conditions). The three types of required conditions are inherited from the requirements realised by the said operation [10–13].

Now, the concept of *agent* stands for an active component in the system (software or environment). Agents may be related to goals in two ways. A first relation is between goals and the agents that are *responsible* for them. This relation is present in languages such as KAOS, TROPOS [3], *i** [18], ALBERT and ALBERT II [8, 9]. Furthermore, TROPOS and *i** add a second relation between goals and the agents *aiming* for them. This consideration helps at determining why each goal appears in a model.

The concept of agent itself has given rise to different characterisations. First comes a description of the agents that are "provided". In this article, we call *actor* such an agent. Actors have *capabilities*, *i.e.*, ways to influence the evolution of the system. Capabilities are commonly considered in the literature, even in methods that do not distinguish several characterisations of agents, such as KAOS. Actors also pursue their own relative goals. This relation between actors and goals raises a social dimension: actors are described as having intentions and interactions with each others. These relations, sometimes dubbed *distributed intentionality*, are at the core of *i** [17, 18] and are also present in TROPOS [3].

The second characterisation emphasises "required" agents, identified with the set of operations they perform. A relation of assignment between roles and actors makes the latter, through the roles they play, *responsible* for the operations in these roles. For instance, agents in KAOS enjoy a responsibility relation w.r.t. goals, and they also come with capabilities (in the form of monitored and controlled variables). The responsibilities of agents are also central in ALBERT [8] and ALBERT II [9]. The latter characterises them with a set of constrained behavioural goals they are responsible for. TROPOS and affiliated literature propose a concept of role for agents seen as sets of actions executed along the run of the system[3, 16].

So, actors aim for goals. But they are also responsible for goals, through the roles they play. And then a question arises, which we call the *assignment problem*: are the actors in charge of a role able to fulfill it?

Several techniques have been proposed to solve this question, notably through the introduction of commitments between agents [5–7, 14]. In this setting, actors commit themselves to other actors for realising certain actions. The question is whether each actor is able to support its role, which is identified with a set of commitments. It is tackled by formalising agents capabilities and commitments in a *propositional* language. We feel this use of a propositional language is limiting in the purpose of formalising the relations between roles and actors' capabilities. Indeed, it does not treat either the precedence relations between operations in the system, which determine the interactions between actors, nor the very existence of these actors and their effective actions upon the system. Actors are only taken into account and described in the informal part of this language.

Thus, some methods in RE offer means for describing precedence among operations in the system and others for describing interacting, social actors. We think that there is a strong interest in combining both approaches in a single language with a semantics including the succession between operations, actors' capabilities and their interactions.

In this paper, we define Khi, a core modelling language, to deal with the assignment problem while retaining the behavioural semantics present in many propositions, especially in Kaos. The formal semantics of Khi is expressed in terms of a multi-agent temporal logic $\mathrm{ATL}_{\mathrm{Khi}}$, a fragment of the decidable logic ATL*. Verifying the correctness of a given assignment then reduces to the satisfaction of an $\mathrm{ATL}_{\mathrm{Khi}}$ formula.

To the best of our knowledge, this is the first proposition that reconciles goal- and agent- oriented RE languages in a formal framework with precedence and multi-agent expression.

This paper is organised as follows. In Sect. 2, we present the metamodel of the Khi language, illustrated with a toy example featuring an online shopping marketplace. In Sect. 3, the semantics of Khi is given in terms of $\mathrm{ATL}_{\mathrm{Khi}}$ formulas. In particular, the assignment problem is formalised. In Sect. 4, we discuss our framework and elaborate on our future work in Sect. 5.

2 The Khi Language

This section gives and comments the different elements and relations in language Khi.

2.1 The Metamodel

We first give a brief overview of the metamodel for language Khi (see Fig. 1) before illustrating its concepts in more details on our toy example.

Actors are available agents. They aim for goals that are structured through refinements. Goals may not all be described formally but this work only considers those goals that are formalisable using LTL. Leaf goals of this sort are then realised into operations. Operations are identified with their effect, described using domain pre- and post-conditions. Their scope of application is also constrained by required pre-, post- and trigger conditions. Then, operations are gathered into roles that may be seen as specifications of "required" agents.

Actors also have capabilities. Each capability is described with a pre-condition and a *window* characterising the action it enables (and explained in Sect. 2.3).

Finally, actors can be gathered into coalitions, which are then assigned roles. The effective ability of a coalition to play the roles it is assigned is determined by

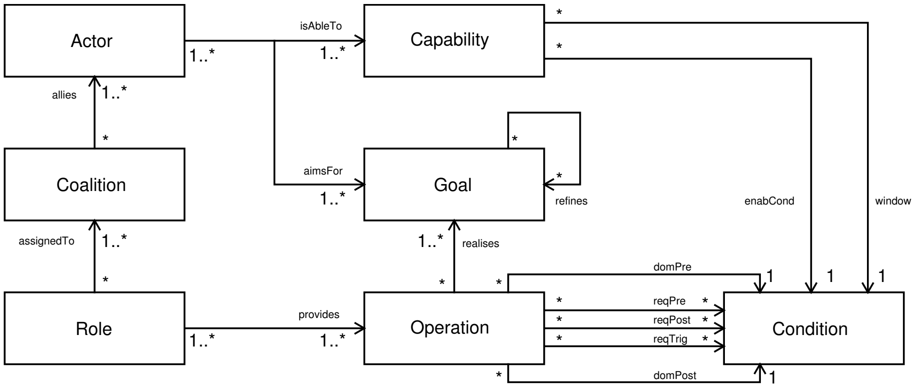


Fig. 1. Metamodel of language Khi

the capabilities of actors it gathers. Considering coalitions enables to consider agents interactions and to express that many actors may cooperate to play a role.

Note that we do not deal with domain properties or OR-refinements in this paper as it is not needed for the presentation of our formal framework. This could be added later and would be dealt with in a similar way as it is in Kaos.

2.2 Actors and Goals

We give in Fig. 2 the goal model in language Khi for our case study. It concerns an online shopping website and actors interacting with it. As in Tropos, actors are represented as dashed ellipses labelled in circles. An ellipse contains the set of goals the enclosing actor aims for. Goals are represented by trapezoids. They are progressively refined into sub-goals. The graphical representation for goals and their refinements is the same as in Kaos.

For instance, the goal *efficientSeelingProcess* for the website is refined into sub-goals *meetSupplyAndDemand* and *simplicityOfTransaction*.

Table 1. Specifications of operations in role *seller*

commitToDeliverGood	$domPre := \top$ ("true") $domPost := cD < 25$ $reqTrig\ for\ instantPayment := s.o = b.o$ $reqPost\ for\ thriftyDeliveryDelay := cD < 12$ $reqPost\ for\ impatientDeliveryDelay := cD < 7$
publishAd	$domPre := \top$ $domPost := s.o < 30$ $reqTrig\ for\ thriftyInterestingPrices := \top$ $reqPost\ for\ thriftyInterestingPrices := s.o < 12$ $reqPost\ for\ impatientInterestingPrices := s.o < 20$

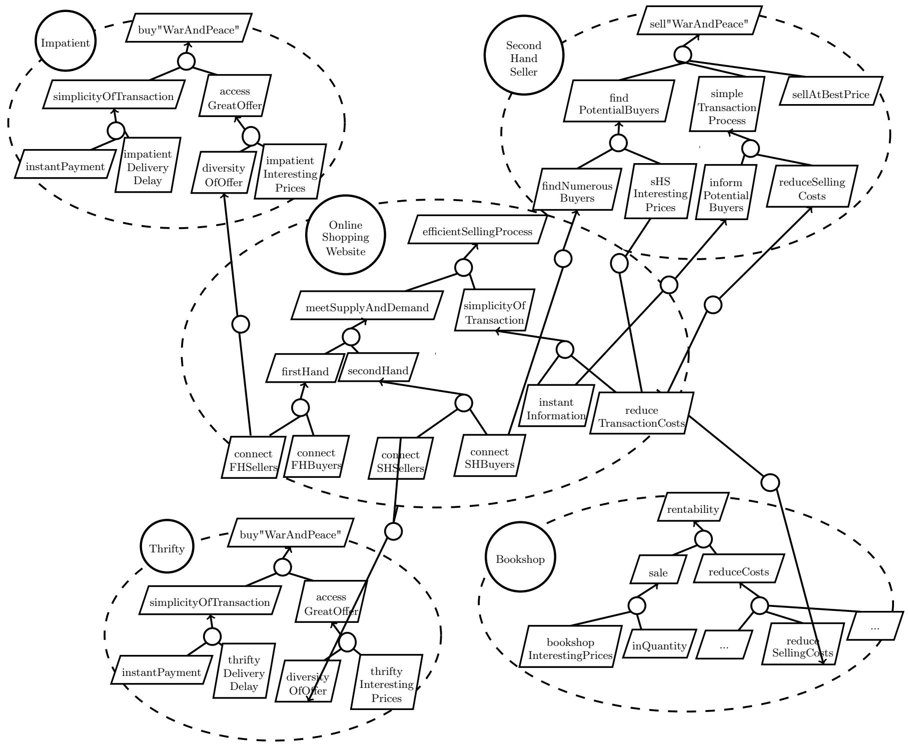


Fig. 2. Actors and goals for the online shopping marketplace

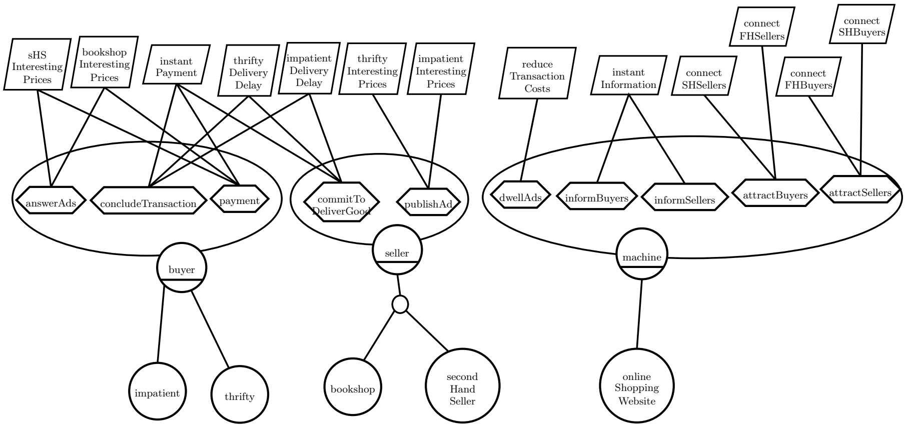


Fig. 3. Leaf goals, operations, roles and assignments for the online shopping marketplace

Table 2. Capabilities of the bookshop and the second hand seller

bookshop	*proposePrice*	*enabCond*:= $\top$ *window* := $s.o \in [15, 30]$
	expedition	*enabCond*:= $s.o = b.o$ *window* := $cD \in [4, 25]$
secondHandSeller	*proposePrice*	*enabCond*:= $\top$ *window* := $s.o \in [9, 20])$
	expedition	*enabCond*:= $s.o = b.o$ *window* := $cD \in [11, 25]$

There are two potential buyers, a *thrifty* one and an *impatient* one. Each of them holds its own goal model. They both want to buy the novel *War and Peace*. Their goal models only differ by the specifications issued from their respective goals *deliveryDelay* and *interestingPrices*, given in Table 1. The *impatient* wants the novels to be delivered within 7 days but may pay for it up to 20 €; while *thrifty* accepts to wait up to 12 days but is unwilling to pay more than 12 € for it. Two different sellers, a bookshop and a second hand seller, having their own goals to satisfy, propose their goods.

The leaf goals in a model are realised by operations to perform. Operations appear in hexagons in Fig. 3. The concept of operations used in KHI follows the definition given for operations in KAOS: they are identified each by a *domPre* and a *domPost* condition. And their executions are constrained by *reqPre*, *reqPost* and *reqTrig* conditions. Thus, as appears in Table 1, *committedDelay*$<$*12* is a *reqPost* for goal *thriftyDeliveryDelay* and *committedDelay*$<$*7* is a *reqPost* for goal *impatientDeliveryDelay*. The action of giving such specifications of operations in order to realise goals is called *operationalisation*.

2.3 Actors and Capabilities

As will be seen in Sect. 3.1, conditions are formalised as formulas in a simple language featuring variables. An actor having capability (*enabCond*, *window*), when *enabCond* holds, can give to variables appearing in *window* any value satisfying *window*. The declaration of the capabilities should be exhaustive: any action an agent is able to perform upon the system is encoded in its table of capabilities. This notion has two main particularities:

1. The capability of an actor is conditioned by the state of the system, *i.e.* by its enabling condition. For instance, a *buyer* can engage a transaction with a *seller* provided the price for the concerned good is in a certain range of values.
2. Then it is possible to model an actor that does not control a variable in its full range. The actor can only give this variable a certain set of values bounded by the window. It is then possible to distinguish the performances of different actors. For instance, a seller controls the delay he can ensure for the delivery to a potential client, but only within a certain window. In

our example, the second hand seller can ensure the delivery in every term between 11 and 25 days, whereas the bookshop can ensure it in every term between 4 and 25 days.

For instance, let us consider operation *commitToDeliverGood* and actor *bookshop*. The conditions of the operation are given by Table 1 and the capabilities of *bookshop* are in Table 2. In these tables and in the following we use the following abbreviations to designate the variables: $s.o$, $b.o$ and cD stand respectively for *seller.offer*, *buyer.offer* and *committedDelay*. The window $cD \in [4, 25]$ enables the bookshop to satisfy the post condition for *impatientDeliveryDelay*: $cD < 7$. But if *window* was interpreted as a classical post-condition, its validity would not ensure the satisfaction of $cD < 7$. Indeed, the post-condition would still be satisfied if, say, $cD = 12$.

Notice that the case where two actors have capabilities with overlapping enabling conditions and (at least) one common variable in their windows leads to a potential deadlock. We call such situation a case of *competing capabilities.* As we are dealing with potential choices and not effective ones, treating effective deadlocks is out of the scope of this paper.

2.4 Assignment of Roles to Coalitions

Operations appear as hexagons in Fig. 3 under leaf goals, and are gathered into roles (plain ellipses labelled by underlined circles). Notice that the methodological question of how this gathering is made by engineers is out of the scope of this paper.

Roles draw a notion of required agents, emerging from the goals specifications. And they are finally assigned to coalitions of actors.

Several coalitions can also be assigned the same role, as is the case for role *buyer* and actors *impatient* and *thrifty*. Note there are two different ways to compose actors:

1. They may play the role together. In our example, *bookshop* and *secondHandSeller* (*sHS*) do not have capabilities to play autonomously the role *seller*. But they can play it together as a coalition. Considering the specifications of role *seller* (Table 1) on the one hand, and the capabilities of actors *bookshop* and *sHS* on the other hand, we deduce that neither actor is able to play fully role *seller*. Indeed, the bookshop cannot put its offer under 15 € and then cannot satisfy the *reqPost* condition for *thriftyInterestingPrice*: $s.o < 12$. And sHS cannot commit to deliver the good under a delay of 11 days and thus cannot satisfy the impatient condition for being delivered within 7 days. But they gather together all the required capabilities for playing the role *buyer*, which is presented with further details in Sect. 3.3.
2. And each of them may play the whole role, just as *impatient* and *thrifty* independently are able to play the role *buyer* on their own. In this case, both *thrifty* and *impatient* are unary coalitions and, as such, are separately assigned the role.

Obviously, a major criterion for the correctness of a model is what we call the assignment problem, that is the question of whether coalitions can play the roles they are assigned.

The verification procedure for the assignment problem enables the requirement engineer to identify the potential lack of capable actors. A role (or parts of roles) that is not assignable to any pre-existing actor identifies one or more fresh actors that should be introduced in the system to satisfy all goals.

3 Semantics

In this section, we give the semantics of language KHI in a logic we call $\mathrm{ATL}_{\mathrm{KHI}}$. To do so, we need the presentation of the following formal background.

3.1 Formal Background: Temporal and Multi-agent Logics

$\mathrm{ATL}_{\mathrm{KHI}}$ is built by integrating temporal and choice operators with propositional logic. It is a fragment of the better known ATL* [2] and thus inherits its decidability.

- A language $\mathrm{Cond}_{\mathrm{KHI}}$, expressing boolean combinations of variable constraints for the description of conditions: *dom* and *req* conditions for operations as well as *enabCond* and *window* conditions for the actors' capabilities.
- The Linear Temporal Logic ($\mathrm{LTL}_{\mathrm{KHI}}$) with atoms in $\mathrm{Cond}_{\mathrm{KHI}}$ for goal expression and operationalisation.
- A multi-agent logic for the mention of actors and the relation of assignment: Alternating-time Temporal Logic for KHI ($\mathrm{ATL}_{\mathrm{KHI}}$). Notably, it will enable us to express the property of a coalition of actors to support a given role.

The expression of conditions, $\mathbf{Cond}_{\mathbf{KHI}}$. Every condition in KHI is described in the language $\mathrm{Cond}_{\mathrm{KHI}}$, a propositional logic which atoms are comparisons of values between variables and natural numbers.

Definition 1. *Given a set of variables U, the language of $\mathit{Cond}_{\mathrm{KHI}}$ over U is given by the following syntax:*

$$\varphi ::= x \sim n \mid x - y \sim n \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi$$

where $x, y \in U$, $n \in \mathbb{N}$, and $\sim \in \{<, >, =, \leq, \geq\}$.

The definition of a *window* uses a fragment of $\mathrm{Cond}_{\mathrm{KHI}}$, denoted by $\mathrm{Cond}^{win}_{\mathrm{KHI}}$, in which variables are explicitly bounded.

Definition 2 ($\mathbf{Cond}^{win}_{\mathbf{KHI}}$). *$\mathit{Cond}^{win}_{\mathrm{KHI}}$ is given by the following grammar:*

$$\varphi ::= a \leq x \wedge x \leq b \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$

where $x \in U$ and $a, b \in \mathbb{N}$.

The simulation of time, LTL [15]. LTL is a temporal logic in which we reason about discrete flow of time. The temporal operators used in LTL are $\circ$ and $\mathbf{U}$. Their intuitive meaning is as follows:

- $\circ\varphi$ expresses that condition φ holds in the next state from the current one.
- $\varphi_1\mathbf{U}\varphi_2$ expresses that the condition φ_1 holds in the current state and remains true until condition φ_2 holds.

Definition 3. *The language of LTL is defined by the following syntax:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \circ\varphi \mid \varphi\mathbf{U}\varphi$$

where p ranges over a countable set P of atomic propositions. In the frame of this article the set of atomic propositions is the set of formulas in the language Cond_{KHI} and we call LTL_{KHI} that instance of LTL.

It is interpreted in ($\mathbb{N}$, V) where $\mathbb{N}$ is the set of natural numbers and $V : \mathbb{N} \to 2^P$ is a function associating each n in $\mathbb{N}$ with a subset of P. For each $n \in \mathbb{N}$, $\mathrm{V}(n)$ is the set of atomic propositions that hold in n.

The relation of satisfaction for LTL is as follows:

Definition 4 (Semantical satisfaction for LTL). *Given an interpretation function $V : \mathbb{N} \to 2^P$ and an integer $i \in \mathbb{N}$, we define the satisfaction relation by induction on the formulas:*

- $V, i \models_{LTL} p$, *for all* $p \in P$, *iff* $p \in \mathrm{V}(i)$.
- $V, i \models_{LTL} \neg\varphi$ *iff* $V, i \not\models \varphi$ *(iff it is not the case that* $V, i \models_{LTL} \varphi$*).*
- $V, i \models_{LTL} \varphi_1 \wedge \varphi_2$ *iff* $V, i \models_{LTL} \varphi_1$ *and* $V, i \models_{LTL} \varphi_2$.
- $V, i \models_{LTL} \circ\varphi$ *iff* $V, i+1 \models_{LTL} \varphi$.
- $V, i \models_{LTL} \varphi_1\mathbf{U}\varphi_2$ *iff* $\exists j \in V(V, i+j \models_{LTL} \varphi_2$ *and* $\forall k < j(V, i+k \models_{LTL} \varphi_1))$.

We use the symbol $\Box$ as an abbreviation meaning that the formula in its scope holds at any time during the execution: $\Box\varphi := \neg(\top\mathbf{U}\neg\varphi)$

Introducing agents in the system, $\mathbf{ATL_{KHI}}$. The semantics of LTL is based on a linear structure, that represents a well-determined evolution of time. In multi-agent logics we consider different possible evolutions of time. At any point in the execution, several potential evolutions are taken into account so that time has a tree-like structure. An important issue when formalising KHI is the expression of the ability of actors to ensure the satisfaction of a given LTL_{KHI} formula ψ. In other words, we need to express the ability of actors to restrict the set of potential executions so that each execution satisfies ψ. This is represented by the introduction of the operator $\langle\langle a\rangle\rangle$, where $\langle\langle a\rangle\rangle\psi$ intuitively means that the agent or coalition (*i.e.*, set) of agents a can ensure the satisfaction of ψ. This formal operation gives the language ATL_{KHI}.

ATL_{KHI} is a fragment of the larger ATL*[1]. It is decidable for the model-cheking and validity problems.

[1] Note that a dual operator $[\![a]\!]$ is often presented in the grammar of ATL*. It is translatable by: $[\![a]\!]\varphi$ iff $\neg\langle\langle a\rangle\rangle\neg\varphi$. We do not need it in our article and keep the notation $[\![]\!]$ for the expression of the semantics.

Definition 5. *The set of ATL_{KHI} formulas is given by the following grammar:*

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle a \rangle\rangle \psi$$

where p is an atomic proposition (cf Definition 3), a is a (coalition of) actor(s) and ψ is a formula in LTL_{KHI}

The operator $\langle\langle a \rangle\rangle$ acts as a complex quantifier over the set of executions: $\langle\langle a \rangle\rangle \varphi$ is true if and only if there are choices for a such that φ is true in all the executions that are compatible with these choices.

The semantics for ATL_{KHI} is given in so-called Concurrent Game Structures (CGS).

– Each state has up to countably many different successors.
– In each state, each agent has a set of available choices.
– There is a transition function: $f : S \times Ch \to S$ where S is the set of states and Ch is the set of possible choices for the different agents. It determines the transitions from each state to its successor.
– An interpretation function V associates each state with a set of atomic propositions.

Definition 6. *Given a CGS M and an actor a in it, we call a strategy for a a function that maps every finite sequence of states in M ending by state s to a choice available for a in s.*

Thus a strategy for an actor gives it a choice for every situation occurring in the execution. Let us now define the satisfaction relation.

Definition 7 (Semantical satisfaction for $\mathbf{ATL_{KHI}}$). *Let M be a CGS, s a state in it, a a coalition of actors, ψ a formula in LTL_{KHI} and φ_1, φ_2 formulas in ATL_{KHI}. Then:*

– *$M, s \models_{ATL_{\text{KHI}}} p$ iff $p \in V(s)$*
– *$M, s \models_{ATL_{\text{KHI}}} \neg\varphi$ iff it is not the case that $M, s \models_{ATL_{\text{KHI}}} \varphi$*
– *$M, s \models_{ATL_{\text{KHI}}} \varphi_1 \wedge \varphi_2$ iff $M, s \models_{ATL_{\text{KHI}}} \varphi_1$ and $M, s \models_{ATL_{\text{KHI}}} \varphi_2$*
– *$M, s \models_{ATL_{\text{KHI}}} \langle\langle a \rangle\rangle \psi$ iff there is a set of strategies F_a, one for each agent in a, such that any execution $\sigma = s_0, s_1, \ldots$ starting from s ($s_0 = s$) and compatible with F_a satisfies ψ ($\sigma, s_0 \models_{LTL} \psi$)[2].*

3.2 Semantics of Khi in $\mathbf{ATL_{KHI}}$

We can now give the semantics of Khi in terms of LTL_{KHI} and ATL_{KHI} formulas[3].

First, a goal g is expressed by a formula in LTL_{KHI}: $[\![g]\!] \in \text{LTL}_{\text{KHI}}$.

[2] ψ is an LTL_{KHI} formula, and is therefore interpreted on an execution, at a certain state of the execution.

[3] ATL_{KHI} is not comparable with ATL, the well-known fragment of ATL*, because the first contains formulas for semantics of role assignment of type $\langle\langle a \rangle\rangle \Box \varphi$, where φ is conjunction of formulas for *req* conditions, which are not expressible into ATL. On the other hand, ATL allows chains of nested choice operators, which are not expressible in ATL_{KHI}.

Definition 8 (Semantics of refines). *The refines relation is such that the satisfaction of the set of refining goals ensures the satisfaction of the refined goals. Let g be a goal*[4]*, then* $\{[\![g.\mathit{refines}^{-1}]\!]\} \models_{LTL_{\text{KHI}}} [\![g]\!]$.

We follow KAOS for the formal semantics of operations and their specifications (recall that domain conditions describe the effects of an operation while required conditions are derived from the goals implemented by the operation).

Definition 9 (Semantics of operation). *An operation op is defined by an occurrence of op.domPre immediately followed by an occurrence of op.domPost.*

$$[\![op]\!] := op.domPre \wedge \circ op.domPost$$

Definition 10 (Semantics of required conditions). *Let op be an operation, then:*

$$\begin{aligned} [\![op.reqPre]\!] &:= \Box([\![op]\!] \rightarrow op.reqPre) \\ [\![op.reqPost]\!] &:= \Box([\![op]\!] \rightarrow \circ op.reqPost) \\ [\![op.reqTrig]\!] &:= \Box((op.domPre \wedge op.reqTrig) \rightarrow [\![op]\!]) \end{aligned}$$

Definition 11 (Semantics of realises). *The realises relation is such that the satisfaction, at every state, of all the specifications of the realising operations, ensures the satisfaction of the realised goal. Let g be a goal, then:*

$$\{[\![g.realises^{-1}.req]\!]\} \models_{LTL_{\text{KHI}}} [\![g]\!]$$

where $g.realises^{-1}.req$ *stands for every conditions in* $g.realise^{-1}.reqPre$, $g.realises^{-1}.reqPost$ *or* $g.realises^{-1}.reqTrig$.

We now come to the proper elements in language KHI, which concerns the characterisation of both concepts of agents and the relation of support that links them together.

First come the definitions of our concepts of agents. They differ by their status in the translation. A *role* is an abstract entity. Its semantics is defined in terms of the operations it provides.

Definition 12 (Semantics of role). *The semantics of a role rl is the conjunction of all the semantics of conditions for the operations it provides.*

$$[\![rl]\!] := \Box \bigwedge_{r \in rl.provides.req} ([\![r]\!])$$

where rl.provides.req stands for every condition in rl.provides.reqPre, rl.provides.reqPost or rl.provides.reqTrig.

[4] Henceforth in this article,

- we note R^{-1} for the converse of the relation R
- $\Gamma \models_{\text{LTL}_{\text{KHI}}} \varphi$ means that φ is a semantic consequence of Γ, *ie.* $\Gamma \models_{\text{LTL}_{\text{KHI}}} \varphi$ iff any structure satisfying the formulas in Γ also satisfies φ.

For instance, the specifications for operations provided by the role *seller* are derived in $\mathrm{LTL}_{\mathrm{KHI}}$ in Table 3, from their description in Table 1.

Actors are given by $\mathrm{ATL}_{\mathrm{KHI}}$ agents and coalitions by $\mathrm{ATL}_{\mathrm{KHI}}$ coalitions.

Let us give the semantics of *isAbleTo*. It is quite different from a view in which the *enabCond* would simply enable the corresponding actor to satisfy the corresponding *window*. In our formalism, an actor holding (*enabCond*, *window*) as a capability not only can force *window* to hold if *enabCond* does, but in this condition he fully controls the value for some variables within the *window*.

Definition 13 (Semantics of isAbleTo). *Let a be an actor. We give the semantics of one of its capabilities $c \in a.isableTo$, and then of the whole set $a.isableTo$. At any state where $c.enabCond$ holds, a is able to give to the variables in $c.window$ any value satisfying it. Formally, let $(x_1, \ldots x_k)$ be the variables in $c.window$. We call $\overline{c}$ the set of vectors $(a_1, \ldots a_k) \in \mathbb{N}^k$ such that $x_1 = a_1, \ldots, x_k = a_k \models_{ATL_{\mathrm{KHI}}} c.window$. Then*

$$\begin{aligned} [\![c]\!] &:= \bigwedge_{(a_1,\ldots a_k)\in\overline{c}} (\langle\!\langle a\rangle\!\rangle(\Box(c.enabCond \rightarrow \circ(x_1 = a_1 \wedge \ldots \wedge x_k = a_k)))) \\ [\![a.isableTo]\!] &:= \bigwedge_{c\in a.isableTo} [\![c]\!] \end{aligned}$$

Let us now give the semantics of the relation *assignedTo*.

Definition 14 (Semantics of assignedTo). *Let r be a role and $c \in r.assignedTo$ a coalition r is assigned to. Then, given the capabilities of coalition c, c is able to play role r,* i.e.,

$$\{[\![a.isAbleTo]\!] \mid a \in c.allies\} \models_{ATL_{\mathrm{KHI}}} \langle\!\langle c.allies\rangle\!\rangle[\![r]\!]$$

3.3 Application to Our Toy Example

Let us illustrate the definitions and the treatment of the assignment problem with our example. In the following we focus on the two different ways, mentioned in Sect.2.4, to compose actors in the assignment: either by gatering them into a coalition or by assigning them the same role.

– A coalition gathers actors together so that they are assigned a role in solidarity. This is represented by the circle in Fig. 3 between the actors *bookshop* and *sHS* and the role *seller*. Here we sketch the proof of the correction of role *seller* being assigned to coalition {*bookshop*, *sHS*}: *bookshop* and *sHS* can jointly play this role. It consists in providing a diverse offer of goods, with both cheap and quickly deliverable items. Let us also stress that neither

Table 3. Semantics of role *seller*

commitToDeliverGood	$((s.o = b.o) \rightarrow \circ(cD < 7))$
publishAd	$\wedge(\top \rightarrow \circ(s.o < 12))$

bookshop nor *sHS* is able to play the role by himself. Role *seller* is given, as mentioned in Table 3, by the formula

$$\square(((s.o = b.o) \rightarrow \circ(cD < 7)) \wedge (\top \rightarrow (\circ(s.o < 12))))$$

Table 2 shows that the bookshop is not able to propose a price under 15 €: $[\![bookshop.isAbleTo]\!] \nvDash \langle\!\langle bookshop\rangle\!\rangle\square(\circ(s.o < 12))$. Thus:

$$[\![bookshop.isAbleTo]\!] \nvDash_{\mathrm{ATL}_{\mathrm{KHI}}} \langle\!\langle bookshop\rangle\!\rangle[\![seller]\!]$$

In a similar way, sHS cannot commit to deliver its good under 12 days so

$$[\![sHS.isAbleTo]\!] \nvDash_{\mathrm{ATL}_{\mathrm{KHI}}} \langle\!\langle sHS\rangle\!\rangle[\![seller]\!]$$

So neither *bookshop* nor *sHS* is able to play role *seller*: they both fail on supporting one of the conditions. But since each of them is able to support the condition the other fails on, they together can ensure the satisfaction of the whole role. Indeed, *bookshop* can ensure the condition for the delay and *sHS* the condition for the price, *i.e.* we have the following situation:

$$[\![bookshop.isAbleTo]\!] \models_{\mathrm{ATL}_{\mathrm{KHI}}} \langle\!\langle bookshop\rangle\!\rangle\square((s.o = b.o) \rightarrow \circ(cD < 7))$$
$$[\![sHS.isAbleTo]\!] \models_{\mathrm{ATL}_{\mathrm{KHI}}} \langle\!\langle sHS\rangle\!\rangle\square(\top \rightarrow \circ(s.o < 12))$$

Then we have:

$$[\![\{bookshop,\ sHS\}.isAbleTo]\!] \models_{\mathrm{ATL}_{\mathrm{KHI}}} \langle\!\langle bookshop\rangle\!\rangle\square(((s.o = b.o) \rightarrow \circ(cD < 7)) \wedge (\langle\!\langle sHS\rangle\!\rangle\square(\circ(s.o < 12))))$$

which entails $[\![\{bookshop,\ sHS\}.isAbleTo]\!] \models_{\mathrm{ATL}_{\mathrm{KHI}}} \langle\!\langle bookshop, sHS\rangle\!\rangle\square[\![seller]\!]$.

– And several coalitions may be assigned the same role. This is the case in our example, where both unary coalitions *impatient* and *thrifty* are assigned role *buyer*. Here we only formulate the assignment problem relative to role *buyer*. To solve it one must prove that $[\![impatient.isAbleTo]\!] \models_{\mathrm{ATL}_{\mathrm{KHI}}} \langle\!\langle impatient\rangle\!\rangle[\![buyer]\!]$ and that $[\![thrifty.isAbleTo]\!] \models_{\mathrm{ATL}_{\mathrm{KHI}}} \langle\!\langle thrifty\rangle\!\rangle[\![buyer]\!]$.

4 Related Work

Khi gives a proposition of language for treating both the behavioural description of goal satisfaction and the mention of social agents, taken into account with their goals, capabilities and interactions.

Both items have been previously studied, but not in a coherent, semantically-rich, formal framework. Kaos proposes a semantic picture based upon temporal traces: behavioural goals are described as LTL formulas and operations as pre- and post-conditions. The notion of intentional agents is at the core of the *i** methodology. Then, the ability of agents to play roles has recently been analysed, notably in terms of commitments they make [5–7, 14].

Our work aims at unifying these achievements into a single language and enrich the semantics for RE with a temporal multi-agent language.

Furthermore, our present work enables to enrich the semantics for RE with the expression of the assignment problem, *i.e.* the very availability the actors have to support their assigned responsibilities in the description of the system. This gives two distinct perspectives for RE.

The role support itself concerns a common problem in RE which is the attribution of specifications each actor should ensure. In KAOS, it appears as the assignment of goals to agents. Nevertheless KAOS does not give any means for discussing or appreciating this question of the actual ability of the agents to ensure their assigned responsibilities.

Once the goals are refined and gathered into roles or *agent types* in TROPOS also they are assigned to actors [3]. Some extensions of TROPOS tackle the question of checking the assignment of such roles to actors [5–7]. But this checking is formalised in propositional logic. Therefore it ignores the precedence of operations as well as the mention of agents, which is made in an informal meta-language. Precedence nevertheless appears as an essential element in the formalisation of actors' actions and interactions. Actor a_1 may, for instance, be able to realise an operation o provided that a condition c is ensured. In case the said condition c is ensurable by an other actor a_2 then a_1 and a_2 are together able to satisfy c, provided that they coordinate their actions upon the system: a_2 should ensure c before a_1 performs o. Identifying synergies between actors and interdependencies thus calls for this expression of precedence.

5 Conclusion

In this paper, we have proposed a formally-rich language for RE that conciliates:

– A classical description of behavioural goals and of operations in terms of temporal logic.
– Two concepts of agents: actors as required agents and roles as prescribed agents. Actors are characterised by the goals they aim for and by capabilities and roles are characterised by operations they provide.
– A multi-agent semantics integrating both the behavioural dimension of goal satisfaction and the ability of actors to support their assigned roles in a model.

Tackling the assignment problem then gives a correctness criterion for an RE model.

Furthermore, identifying the role support can be used for giving further precisions about the specifications for the software to be introduced in a multi-agent system. KHI brings tools to identify the specified operations to be ensured and, within this set, to sort the readily ensurable ones and the ones to be provided by the software. The lasts are the very specifications of the software itself. This difference at the level of specified operations is similar to a distinction made in KAOS between leaf goals assigned to agents in the environement (*expectations*) and to agents in the software (*requirements*). Treating this distinction at the level of operations, Khi offers tools for distinguishing expected and required operations. Fig.[3], eg, shows that our case study identifies an unique lacking role, called *machine* and gathering the five operations *dwellAds*, *financeAds*, *attractBuyers*, *informBuyers* and *informSellers*.

Concerning our future work, we first plan to develop fully the support for verification of Khi models. Indeed, to the best of our knowledge, there are algorithms but no available tools[5] to check the validity and perform model-checking of $\mathrm{ATL}_{\mathrm{Khi}}$ formulas. This will come with the further study of $\mathrm{ATL}_{\mathrm{Khi}}$ itself. It will in particular enable us to assess our approach concerning the assignment problem and potential deadlocks due to competing capabilities.

In this paper, we presented the semantics of Khi through a translation into $\mathrm{ATL}_{\mathrm{Khi}}$ formulas. In the future, we will directly describe the semantic model of actors and their capabilities in terms of (a fragment of) CGS. A first reason for this is to help build a more intuitive semantic picture of Khi concepts. More technically, some of the verification problems associated to an instance of Khi will then reduce to a model-checking problem (instead of the current semantic consequence problem).

Now, since the assignment problem has been successfully stated (thanks to the formal verification proposed in this paper), it is known whether all roles *can* be played by some available actors. But let us stress that they still might make other choices invalidating their assigned roles.

A natural question is then: how to distinguish between the ability of an actor to play a role and the fact that he will actually play it? A solution to this problem may be to reify actors' strategies [4] and so distinguish between an effective behaviour as a so reified strategy and a capability. A formalism with strategy would indeed enable to check the coherence between the behaviour of an actor and its assigned role. It would also enable the expression of coherence between two behaviours and then to express solutions for avoiding an effective deadlock in the case of a potential one.

References

1. Alur, R., Henzinger, T., Mang, F., Qadeer, S., Rajamani, S., Tasiran, S.: MOCHA: Modularity in model checking. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998)
2. Alur, R., Henzinger, T., Kupferman, O.: Alternating-time temporal logic. J. ACM, 672–713 (2002)
3. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems, 203–236 (2004)
4. Brihaye, T., Da Costa, A., Laroussinie, F., Markey, N.: ATL with strategy contexts and bounded memory. Logical Foundations of Computer Science, 92–106 (2009)

[5] The Mocha tool [1] offers facilities for the verification of ATL formulas, but not $\mathrm{ATL}_{\mathrm{Khi}}$ ones.

5. Chopra, A., Dalpiaz, F., Giorgini, P., Mylopoulos, J.: Modeling and reasoning about service-oriented applications via goals and commitments. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 113–128. Springer, Heidelberg (2010)
6. Chopra, A., Dalpiaz, F., Giorgini, P., Mylopoulos, J.: Reasoning about agents and protocols via goals and commitments. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, vol. 1, pp. 457–464 (2010); International Foundation for Autonomous Agents and Multiagent Systems
7. Chopra, A., Singh, M.: Multiagent commitment alignment. In: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems, vol. 2, pp. 937–944 (2009); International Foundation for Autonomous Agents and Multiagent Systems
8. Du Bois, P.: The Albert II reference manual. Tech. rep., University of Namur, Belgium (1997)
9. Dubois, E., Du Bois, P., Petit, M.: ALBERT: an agent-oriented language for building and eliciting requirements for real-time systems. In: Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences. Information Systems: Collaboration Technology Organizational Systems and Technology, vol. 4, pp. 713–722. IEEE (1994)
10. van Lamsweerde, A.: Requirements engineering, From System Goals to UML Models to Software Specifications. Wiley (2009)
11. Letier, E., van Lamsweerde, A.: Agent-based tactics for goal-oriented requirements elaboration. In: Proceedings of the 24rd International Conference on Software Engineering, ICSE 2002, pp. 83–93 (May 2002)
12. Letier, E., Van Lamsweerde, A.: Deriving operational software specifications from system goals. In: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, p. 128. ACM, New York (2002)
13. Letier, E.: Reasoning about Agents in Goal-Oriented Requirements Engineering. Ph.D. thesis, Universite Catholique de Louvain (November 05, 2002)
14. Mallya, A., Singh, M.: Incorporating commitment protocols into Tropos. In: Müller, J.P., Zambonelli, F. (eds.) AOSE 2005. LNCS, vol. 3950, pp. 69–80. Springer, Heidelberg (2006)
15. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Compouter Science, pp. 46–57 (1977)
16. Silva, C., Castro, J., Tedesco, P., Araújo, J., Moreira, A., Mylopoulos, J.: Improving the architectural design of multi-agent systems: the tropos case. In: Proceedings of the 2006 International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, pp. 107–113. ACM (2006)
17. Yu, E.: Agent-oriented modelling: software versus the world. In: Wooldridge, M.J., Weiß, G., Ciancarini, P. (eds.) AOSE 2001. LNCS, vol. 2222, pp. 206–225. Springer, Heidelberg (2002)
18. Yu, E.: Social modelling and i*. In: Conceptual Modelling: Foundations and Applications (2009)

Author Index