18th Brazilian Symposium on Formal Methods

SBMF 2015

Belo Horizonte, MG, Brazil

September 21-22, 2015

Pre-proceedings

Preface

This volume (pre-proceedings) contains the papers expected to be presented at SBMF 2015: the 18th Brazilian Symposium on Formal Methods. The conference was held in Belo Horizonte, Brazil, from September 21 to September 22 colocated with CBSoft 2015, the 6th Brazilian Conference on Software: Theory and Practice.

The Brazilian Symposium on Formal Methods (SBMF) 2015 is the eighteenth of a series of events devoted to the development, dissemination and use of formal methods for the construction of high-quality computational systems. It is now a well-established event, with an international reputation. The first edition occurred in 1998.

The conference included two invited talks given by Adenilso Simão (ICMC/USP, São Carlos, SP, Brazil) and Sumit Gulwani (Microsoft Research, WA, USA), who also taught at CBSoft tutorial ”Programming by Examples”.

A total of 11 papers were expected to be presented at the conference. They were selected from 25 submissions that came from 10 different countries: Brazil, Canada, France, Germany, Luxembourg, Netherlands, South Africa, Sweden, United Kingdom, and United States of America. The Program Committee was composed by 41 members from the national and international community of formal methods. Each submission was reviewed by at least 3 program committee members. The process of submissions by the authors, paper reviews, and deliberations of the Program Committee were all assisted by EasyChair.

We are grateful to the Program Committee, and to the additional reviewers, for their hard work in evaluating submissions and suggesting improvements. In particular, special thanks go to Christiano Braga, Juliano Iyoda and Rohit Gheyi, co-chairs of previous editions of SBMF, who were always available to help us and to share his experience and wisdom. We are very thankful to the general chairs of CBSoft 2015, Eduardo Figueiredo (UFMG), Fernando Quintão (UFMG), Kecia Ferreira (CEFET-MG), and Maria Augusta Nelson (PUC-MG), who made everything possible for the conference to run smoothly.

SBMF 2015 was organized by Universidade Federal de Minas Gerais, Centro Federal de Educação Tecnológica de Minas Gerais, and Pontifícia Universidade Católica de Minas Gerais, promoted by the Brazilian Computer Society (SBC), and sponsored by the following organizations, who we thank for their generous support: CAPES, CNPq, FAPEMIG, Google, RaroLabs, Take.net, ThoughtWorks, AvenueCode, Avanti Negócios e Tecnologia.

September 2015 — Márcio Cornélio
Belo Horizonte — Bill Roscoe

Table of Contents

Program Committee

Aline Andrade	Universidade Federal da Bahia
Wilkerson L. Andrade	Federal University of Campina Grande
Luis Barbosa	Universidade do Minho
Christiano Braga	Universidade Federal Fluminense
Michael Butler	University of Southampton
Ana Cavalcanti	University of York
Simone André Da Costa Cavalheiro	Universidade Federal de Pelotas
Márcio Cornélio	Centro de Informática - UFPE
Andrea Corradini	Dipartimento di Informatica, Università di Pisa
Jim Davies	University of Oxford
David Deharbe	Universidade Federal do Rio Grande do Norte - Departamento de Informatica e Matematica Aplicada
Ewen Denney	SGT/NASA Ames
Clare Dixon	University of Liverpool
Adalberto Farias	UFCG
Rohit Gheyi	Department of Computing Systems - UFCG
Rolf Hennicker	Ludwig-Maximilians-Universitt Mnchen
Juliano Iyoda	Centro de Informática, Universidade Federal de Pernambuco
Peter Gorm Larsen	Aarhus University
Bruno Lopes	Pontifícia Universidade Católica do Rio de Janeiro
Anamaria M. Moreira	Federal University of Rio de Janeiro
Patricia Machado	Federal University of Campina Grande
Narciso Marti-Oliet	Universidad Complutense de Madrid
Tiago Massoni	Universidade Federal de Campina Grande - UFCG
Ana Melo	University of Sao Paulo
Alvaro Moreira	Federal Univeristy of Rio Grande do Sul
Alexandre Mota	Universidade Federal de Pernambuco
Arnaldo Moura	IC/UNICAMP
Leonardo Moura	Microsoft Research
Peter Müller	ETH Zürich
David Naumann	Stevens Institute of Technology
Jose Oliveira	Universidade do Minho
Marcel Vinicius Medeiros Oliveira	Universidade Federal do Rio Grande do Norte
Leila Ribeiro	Universidade Federal do Rio Grande do Sul
Bill Roscoe	University of Oxford
Augusto Sampaio	Federal university of Pernambuco
Leila Silva	Universidade Federal de Sergipe
Adenilso Simao	ICMC/USP
Sofiene Tahar	Concordia University

Leopoldo Teixeira	Informatics Center, Federal University of Pernambuco
Heike Wehrheim	University of Paderborn
Jim Woodcock	University of York

Additional Reviewers

A

Aguirre, Luis
B

Bonifácio, Adilson
D

Daghar, Alaeddine
H

Hachani, Ahmed
L

Lopes, Bruno
M

Macedo, Nuno
Madeira, Alexandre
P

Pita, Isabel
S

Santiago Júnior, Valdivino
Soares, Gustavo
T

Tavares, Cláudia

Applications of Formal Methods to Data Wrangling and Education

Sumit Gulwani

Microsoft Research, Redmond, WA, USA
sumit@microsoft.com

Abstract. Data is locked up in semi-structured formats such as spreadsheets, text/log files, webpages, pdf documents. Getting data out of these documents into structured formats that allow the data to be explored and analyzed is challenging. While data scientists spend 80% of their time cleaning data, programmatic solutions to data manipulation are beyond the expertise of 99% of end users who do not know programming. Programming by Examples (PBE) can make data wrangling a delightful experience for the masses. The first part of this talk will describe how formal methods can be used to address two key challenges in PBE: (a) efficient search algorithms to explore the huge state space of programs to find those that match the user-provided examples, and (b) effective ambiguity resolution techniques to deal with the inherent ambiguity in the examples. The second part of this talk will describe how formal methods can help automated two key tasks in Education, namely (a) problem generation and (b) feedback generation. I will illustrate this using recent research results that have been applied to various STEM subject domains including mathematics, programming, logic, and automata theory. These results advance the state-of-the-art in intelligent tutoring, and can play a significant role in enabling personalized and interactive education in both standard classrooms and MOOCs.

To Test or not to Test: That is a formal question

Adenilso Simao

Universidade de São Paulo, São Carlos, Brazil
adenilso@icmc.usp.br

Abstract. The demand of highly dependable software has greatly motivated the research of two important areas of software engineering, namely, formal methods and software testing. Both areas have matured considerably in the last years, to the point of being mainstream approaches in the development of critical systems. However, despite some fruitful exchange of ideas between them, formal methods and software testing have advanced somehow isolated from each other. In this talk, we review the achievements related to the combination of formal methods and software testing. We will discuss, for instance, how testing can be formal and how formal methods can be aided by testing. The main goal of this talk is to identify opportunities to strengthen the exchange between these two exciting and important areas.

Keywords: Formal Methods, Software Testing

1 Context

We currently live in the so-called information age; the capacity to gather, treat and share data is higher than ever. More and more tasks are delegated to automated systems. Our dependency on these systems increases accordingly. Thus, it is more important than ever to have systems we can depend on, i.e., systems which provide the correct result (perform the expected tasks, as requires), in a timely manner.

However, the relevant question is how to obtain systems with the required level of quality, considering the constraints on cost and time available to build them. Several techniques and methods having been proposed for tackling this problem. In this talk, we focus on two such techniques, which are complementary to each other in a sense, but have not been integrated enough: formal methods and testing.

On one hand, formal methods employ mathematical techniques to improve the development of high-quality systems. The mathematical background is employed to provide a sound reasoning about the correctness of the artifact produced during the development. The ambiguity, which is common in natural language requirements and informal design techniques, is greatly avoided. Thus, the precise syntax and semantics of the notation are used to solidly build the software. This approach is more optimistic about the potentiality of human capacity (aided by automated tools). There is, however, a semantic gap between the informal, messy "real" world and the formal, neat models. The gap will always be there.

On the other hand, testing techniques exercise the real system with real inputs, to identify possible bugs. It is more of a pragmatic activity, trying to figure out what could go wrong and how to force the system to go wrong, if it could. In a sense, this approach is pessimistic; to do a good job, the tester should doubt everything and trust nobody. Finding the balance out of this paranoiac view is a practical skill, which comes with experience. The research in this area is usually a bit too informal, too common sense.

2 Objective of the Talk

We consider that both approaches start from too opposite ends with the same goal. Somehow, the ideal situation is somewhere in the middle. The mathematical basis of formal methods can aid the testing activities to solve many problems, especially the selection of test inputs and the definition of oracle for test cases. The testing activities can also be included in the formal methods to help to bridge the semantic gap.

In this talk, we show the approaches that already merge formal and testing techniques. We discuss and illustrate, e.g., Model-Based Testing [1], IOCO Theory [4], Formal Testing [2] and Concolic Testing [3]. We then conclude with some open issues, research opportunities and future work.

References

1. Brinksma, E., Grieskamp, W., Tretmans, J. (eds.): Perspectives of Model-Based Testing, 5.-10. September 2004, Dagstuhl Seminar Proceedings, vol. 04371. IBFI, Schloss Dagstuhl, Germany (2005), http://drops.dagstuhl.de/portals/04371/
2. Gaudel, M.C.: Testing can be formal, too. In: International Joint Conference, Theory And Practice of Software Development. Lecture Notes in Computer Science, vol. 915, pp. 82–96. Springer-Verlag (1995)
3. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005. pp. 213–223. ACM (2005), http://doi.acm.org/10.1145/1065010.1065036
4. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)

Towards Reasoning in Dynamic Logics with Rewriting Logic: the Petri-PDL Case

Christiano Braga and Bruno Lopes*

Instituto de Computação,
Universidade Federal Fluminense, Niterói, Brazil
cbraga@ic.uff.br, bruno@ic.uff.br

Abstract. Safety is a desired property in software to ensure that no unforeseen scenarios will be achieved and in concurrent systems the variety of scenarios increase the complexity. Dynamic Logics (DL) present a large body of techniques to reason about systems and certify systems. Modelling and assessing concurrent systems with a formal semantics leads to the possibility of proving that they comply with their specification. Petri nets fulfill the requirements as a formal modelling language comprising a wide body of tools and an intuitive graphical interpretation. Petri-PDL puts together DL with Petri nets, providing a theoretical background to reason about Petri nets, inheriting their properties with all the techniques available for DL. This work presents a prototype implementation, in the Rewriting Logic language Maude, of a bounded model checker for Petri-PDL. The Petri-PDL model checker is formally designed following the representation of Kripke models as rewrite theories defined for the Linear Temporal Logic model checker available in Maude.

1 Introduction

Concurrency is ubiquitous in modern systems and uncertified software may lead to unforeseen scenarios. Formal methods should ensure that systems behave as expected by verifying the presence or absence of given properties. To state that some property is certified corresponds to present a proof of the given property. Proof theory, model theory and their automation play an important role to this end.

Logic is about proofs and proof theory is its branch that deals with proofs as mathematical objects. A program execution may denote a transition in a discrete state system, that is, with satisfied preconditions, a program is executed and a resulting configuration is achieved. Based on this idea, Dynamic Logics [6] present a family of logic systems to reason about programs.

Propositional Dynamic Logic (PDL, [5]) combines classical Propositional Logic with Dynamic Logics, a family of modal logics where each program corresponds to a modality [6]. The PDL formula $\langle\alpha\rangle p$ denotes that a propositional

* The author thanks to CNPq, CAPES and FAPERJ for partially supporting this work.

property p holds after some execution of a program α. PDL may be extended in a "logic engineering" way. Each α-language (the language for α programs) gives rise to a different Dynamic Logic. Let us keep that in mind for a moment.

Petri nets are an expressive and widely used formalism to specify concurrent systems [2,12,15]. They offer a formal semantics with an intuitive graphical interpretation. It leads not only to precision to model but also to easiness and readiness.

Returning our focus to PDL, in [9] the second author replaced the program α in PDL with a marked Petri net "s, π", where s corresponds to the marking of the Petri net π, giving rise to Petri-PDL. This new PDL is a sound, complete and decidable [9,11] logic system to reason about Petri nets. Moreover, in other Dynamic Logic-based approaches to reason about Petri nets, it is sometimes required to translate the net to another language, usually without a bijection between Petri nets and the target language [14]. Petri-PDL represents Petri nets quite naturally without requiring such a translation.

The semantics of Dynamic Logics is usually denoted by a Kripke frame [6], i.e., a tuple with a set of worlds (states) and a relation among these worlds. Given a frame $\mathcal{F}$ and a property φ, to verify if φ holds in a state w from $\mathcal{F}$ is to verify the satisfaction relation $\mathcal{F}$,$w \Vdash \varphi$. Therefore, to verify if φ holds, it is necessary to look for a path on the structure induced by $\mathcal{F}$ beginning in w that satisfy φ. Such a relation may be naturally represented as a rewrite relation in a suitable term-rewriting system.

The main contribution of this work is a prototype implementation of a bounded model checker for Propositional Dynamic Logics (PDL). These are the first steps towards a framework to implement model checkers for Dynamic Logics in Maude. The prototype now focuses on Petri-PDL and by associating a rewrite theory in Maude to a given Petri-PDL net.

Plan of the paper. Section 2 recalls basic definitions for Propositional Dynamic Logic. In Section 3 we briefly discuss Rewriting Logic and the Maude language. Section 4 shows how we can apply the mapping from Kripke structures to rewrite theories in order to represent PDL as rewrite theories. Section 5 shows how PDL is extended to allow the specification and reasoning of Petri nets. Section 6 is the main contribution of the paper discussing a prototype implementation of a model checker for Petri-PDL in Maude, exploring the metaprogramming facilities of Rewriting Logic and Maude. Section 7 acknowledges some related work and finally Section 8 concludes this paper.

2 Propositional Dynamic Logic

The usual approach to Dynamic Logic is reflected in Propositional Dynamic Logic (PDL) [5,6]. This section presents its syntax and semantics.

Definition 1 (PDL language). *The PDL language consists of a set Φ of countably many proposition symbols, a set Π of countably many basic programs, the boolean connectives $\neg$ and $\wedge$, the program constructors $;$, $\cup$ and * and a* *modality $\langle\pi\rangle$ for every program π. Let $p \in \Phi$ and $a \in \Pi$, formulas are defined as*

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle\pi\rangle\varphi, \text{ with } \pi ::= a \mid \pi;\pi \mid \pi \cup \pi \mid \pi^\star.$$

The standard abbreviations are valid: $\bot \equiv \neg\top$, $\varphi \vee \phi \equiv \neg(\neg\varphi \wedge \neg\phi)$, $\varphi \to \phi \equiv \neg(\varphi \wedge \neg\phi)$ and $[\pi]\varphi \equiv \neg\langle\pi\rangle\neg\varphi$.

Definition 2 (PDL frame). *A* frame *for PDL is a tuple $\mathcal{F} = \langle W, R_\pi\rangle$ where*

- *W is a non-empty set of states;*
- *R_π is a binary relation over W, for each basic program $\pi \in \Pi$;*
- *We can inductively define a binary relation R_π, for each non-basic program π, as follows*
  - $R_{\pi_1;\pi_2} = R_{\pi_1} \circ R_{\pi_2}$,
  - $R_{\pi_1 \cup \pi_2} = R_{\pi_1} \cup R_{\pi_2}$,
  - $R_{\pi^\star} = R_\pi^\star$, *where $R_\pi^\star$ denotes the reflexive transitive closure of R_π.*

Definition 3 (PDL model). *A* model *for PDL is a pair $\mathcal{M} = \langle\mathcal{F}, \mathbf{V}\rangle$, where $\mathcal{F}$ is a PDL frame and $\mathbf{V}$ is a valuation function $\mathbf{V} : \Phi \to 2^W$.*

Definition 4 (PDL satisfaction notion). *Let $\mathcal{M} = \langle\mathcal{F}, \mathbf{V}\rangle$ be a model. The notion of* satisfaction *of a formula φ in a model $\mathcal{M}$ at a world w, notation $\mathcal{M}, w \Vdash \varphi$, can be inductively defined as follows:*

- *$\mathcal{M}, w \Vdash p$ iff $w \in \mathbf{V}(p)$;*
- *$\mathcal{M}, w \Vdash \top$ always;*
- *$\mathcal{M}, w \Vdash \neg\varphi$ iff $\mathcal{M}, w \nVdash \varphi$;*
- *$\mathcal{M}, w \Vdash \varphi_1 \wedge \varphi_2$ iff $\mathcal{M}, w \Vdash \varphi_1$ and $\mathcal{M}, w \Vdash \varphi_2$;*
- *$\mathcal{M}, w \Vdash \langle\pi\rangle\varphi$ iff there is $w' \in W$ such that $wR_\pi w'$ and $\mathcal{M}, w' \Vdash \varphi$.*

3 Rewriting logic

In this Section we recall the logical foundations of Maude. Rewriting logic, Maude's underlying logical framework, is parameterized by an equational logic. In Section 3.1 we discuss membership equational logic [1], a generalization of equational logic. Section 3.2 discusses Rewriting Logic. Section 3.3 discusses Maude, a concrete syntax for Rewriting Logic.

3.1 Membership equational logic

Membership equational logic (MEL) [1] is an expressive version of equational logic. MEL supports sorts, subsorts, and operator overloading. Errors and partiality are specified through *kinds* and conditional membership axioms.

A signature in membership equational logic is a triple (K, Σ, S) (just Σ in the following), with K a set of kinds, $\Sigma = \{\Sigma_{k_1\ldots k_n,k}\}_{(k_1\ldots k_n,k)\in K^*\times K}$ a many-kinded signature, and $S = \{S_k\}k \in K$ a pairwise disjoint K-kinded family of sets of sorts. The kind of a sort s is denoted by $[s]$. Notation $T_{\Sigma,k}$ and $T_{\Sigma,k}(\overrightarrow{x})$

denotes respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in $\overrightarrow{x}$, where $\overrightarrow{x} = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of K-kinded variables. The notation $t(\overrightarrow{x})$ makes explicit the set of variables that appear in the term t. Atomic formulas in MEL are either equations $t = t$, where t and t are terms of the same kind, or membership assertions of the form $t : s$, where the term t has kind k and $s \in S_k$. Sentences are Horn clauses on these atomic formulas, i.e., sentences of the form $(\forall \overrightarrow{x})A_0$ *if* $A_1 \wedge \ldots \wedge A_n$, where each A_i is either an equation or a membership assertion, and $\overrightarrow{x}$ is a set of K-kinded variables that contains all the variables occurring in $A_0, A_1, \ldots, A_n$.

A theory in membership equational logic is a pair (Σ, E), where E is a finite set of sentences in membership equational logic over the signature Σ. Properly typed terms are those that can be proved to have a sort. If there is not such a proof then a term is ill-sorted: it has a kind but not a sort. For example, assuming sum $+$, difference $-$ and integer division $/$ operators with the appropriate declarations, $3 + 2 : \mathit{Nat}$ and $34 : \mathit{Int}$, but $7/0$ is a term of kind $[\mathit{Int}]$ with no sort. Fig. 1 presents the calculus for MEL, where θ is the substitution function.

$$\frac{t \in T_\Sigma(\overrightarrow{x})}{(\forall \overrightarrow{x})t = t}\ \textit{Reflexivity} \qquad \frac{(\forall \overrightarrow{x})t' : s \quad (\forall \overrightarrow{x})t = t'}{(\forall \overrightarrow{x})t : s}\ \textit{Membership}$$

$$\frac{(\forall \overrightarrow{x})t' = t}{(\forall \overrightarrow{x})t = t'}\ \textit{Symmetry} \qquad \frac{(\forall \overrightarrow{x})t_1 = t_2 \quad (\forall \overrightarrow{x})t_2 = t_3}{(\forall \overrightarrow{x})t_1 = t_3}\ \textit{Transitivity}$$

$$\frac{f \in \Sigma_{k_1 \ldots k_n, k}(\forall \overrightarrow{x})t_i = t'_i \quad t_i, t'_i \in T_{\Sigma, k_i}(\forall \overrightarrow{x}) \quad 1 \leq i \leq n}{(\forall \overrightarrow{x})f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)}\ \textit{Congruence}$$

$$\frac{\begin{array}{c}(\forall \overrightarrow{x})A_0 \ \mathit{if}\ A_1 \wedge \ldots \wedge A_n \in E \\ \theta : \overrightarrow{x} \to T_\Sigma(\overrightarrow{y}) \quad (\forall \overrightarrow{y})\theta(A_i)\ 1 \leq i \leq n\end{array}}{(\forall \overrightarrow{y})\theta(A_0)}\ \textit{Replacement}$$

Fig. 1. Deduction rules for membership equational logic.

A theory (Σ, E) in MEL has an initial model, denoted $T_{\Sigma/E}$, whose elements are equivalence classes $[t]_E$ of ground terms. In the initial model, sorts are interpreted as the smallest sets satisfying the axioms in the theory, and equality is interpreted as the smallest congruence satisfying those axioms.

3.2 Rewriting logic

Concurrent systems are axiomatized in Rewriting Logic by means of rewrite theories of the form $\mathcal{R} = (\Sigma, E, R)$. The set of states is described by a membership equational theory (Σ, E) as the algebraic data type $T_{\Sigma/E,k}$ associated to the initial algebra $T_{\Sigma/E}$ of (Σ, E) by the choice of a kind k of states in Σ. The system's transitions are axiomatized by the conditional rewrite rules R which are of the general form

$$\lambda : (\forall \overrightarrow{x})t \to t' \;\; \mathit{if} \;\; \bigwedge_{i\in I} p_i = q_i \wedge \bigwedge_{j\in J} w_j : s_j \wedge \bigwedge_{l\in L} t_l \to t'_l,$$

with λ a label, $p_i = q_i$ and $w_j : s_j$ atomic formulas in MEL for $i \in I$ and $j \in J$, and for appropriate kinds k and $k_l, t, t' \in T_{\Sigma,k}(\overrightarrow{x})$, and $t_l, t'_l \in T_{\Sigma,k_l}(\overrightarrow{x})$ for $l \in L$. Unbounded variables are allowed in the conditions provided that they are added incrementally [3] in terms of variables in the left-hand side of the rule and in the conditions as well.

Deduction rules for Rewriting Logic are given in Fig. 2. Given two states $[u], [v] \in T_{\Sigma/E,k}$, $[v]$ can be reached from $[u]$ by some possibly complex concurrent computation iff it can proven that $u \longrightarrow v$ in the logic.

$$\frac{t \in T_\Sigma(\overrightarrow{x})}{(\forall \overrightarrow{x})t \longrightarrow t}\;\mathit{Reflexivity} \qquad \frac{(\forall \overrightarrow{x})t_1 \longrightarrow t_2 \quad (\forall \overrightarrow{x})t_2 \longrightarrow t_3}{(\forall \overrightarrow{x})t_1 \longrightarrow t_3}\;\mathit{Transitivity}$$

$$\frac{E \vdash (\forall \overrightarrow{x})t = u \quad (\forall \overrightarrow{x})u \longrightarrow u' \quad E \vdash (\forall \overrightarrow{x})u' = t'}{(\forall \overrightarrow{x})t \longrightarrow t'}\;\mathit{Equality}$$

$$\frac{f \in \Sigma_{k_1 \ldots k_n,k} (\forall \overrightarrow{x})t_i \longrightarrow t'_i \quad t_i, t'_i \in T_{\Sigma,k_i}(\overrightarrow{x}) \quad 1 \le i \le n}{(\forall \overrightarrow{x})f(t_1, \ldots, t_n) \longrightarrow f(t'_1, \ldots, t'_n)}\;\mathit{Congruence}$$

$$\frac{\begin{array}{c}(\forall \overrightarrow{x})\lambda : t \to t' \;\mathit{if}\; \bigwedge_{i\in I} p_i = q_i \wedge \bigwedge_{j\in J} w_j : s_j \wedge \bigwedge_{l\in L} t_l \to t'_l \in R \\ \theta : \overrightarrow{x} \to T_\Sigma(\overrightarrow{y}) \quad (\forall \overrightarrow{y})\theta(t_l) \longrightarrow \theta(t'_l) \; l \in L \\ E \vdash (\forall \overrightarrow{y})\theta(p_i) = \theta(q_i) \; i \in I \quad E \vdash (\forall \overrightarrow{y})\theta(w_j) : s_j \; j \in J\end{array}}{(\forall \overrightarrow{y})\theta(t) \longrightarrow \theta(t')}\;\mathit{Replacement}$$

Fig. 2. Deduction rules for Rewriting Logic.

3.3 Maude language

Now that we have discussed MEL and Rewriting Logic, it is easier to talk about Maude as it is a concrete syntax for Rewriting Logic. In order to model a system in Rewriting Logic, that is, to specify such a system in Maude, its static part (state structure) and its dynamics (state transitions) are distinguished. The static part is specified by means of an equational theory (many-sorted, order-sorted or MEL), while the dynamics are specified by means of rules. Computation in a transition system is then precisely captured by the term rewriting relation using those rules, where terms represent states of the given system.

The distinction between the static part and the dynamic part is reflected in Maude by means of functional and system modules. Functional modules in

```
 1 fmod AUTOMATON is
 2   sort Alphabet Alphabet* InitialState FinalState State Configuration .
 3   subsort InitialState FinalState < State .
 4   subsort Alphabet < Alphabet* .
 5   subsort State Alphabet* < Configuration .
 6
 7   op epsilon : → Alphabet* .
 8   op __ : Alphabet* Alphabet* → Alphabet* [assoc id: epsilon] .
 9   op _,_ : Configuration Configuration → Configuration [assoc comm] .
10 endfm
```

Listing 1. AUTOMATON functional module.

Maude correspond to membership equational theories (Σ, E) which are assumed to be Church-Rosser (confluent and sort decreasing) and terminating. Equations are used to define functions over static data as well as properties of states. Usually the equations E is the union of a set A of structural axioms (such as associativity, commutativity, or identity), also known as equational attributes, for which matching algorithms exist in Maude, and a set E' of equations that are Church-Rosser and terminating modulo A.

System modules in Maude correspond to rewrite theories $(\Sigma, A \cup E', R)$ where rewriting with R is performed modulo the equations $A \cup E'$. Moreover, the rules R must be coherent with respect to the equations E' modulo A. Coherence means that the interleaving of rewriting with rules and rewriting with equations will not loose rewrite computations, that is, failing to perform a rewrite that would otherwise have been possible before an equational deduction step was taken. By assuming coherence, Maude always reduces to canonical form using E before applying any rule in R.

A Maude example. In the following example, we illustrate the syntax of Maude modules by representing the computations of a non-deterministic automaton in terms of associative-commutative-identity rewriting. Module AUTOMATON in Listing 1 declares sort Alphabet* for words in an alphabet with an homonymous sort, constructed with an associative and commutative juxtaposition operation (line 8) with identity given by constant empty of sort Alphabet*. Configurations of an automaton are given by the current state of the automaton together with a subword of the input word. Configurations are constructed with an associative comma operator (line 9).

Listing 2 gives an example of a non-deterministic automaton that represents computations of an automaton that accepts words that have “aa” or “bb” as subwords of a given word over the alphabet $\{a, b\}$. After including the functional module AUTOMATON, module AA-BB-SUBWORD declares (in lines 4 and 5) ‘a’ and ‘b’ to be constants of sort Alphabet, and ‘q0’, ‘q1’, ‘q2’, and ‘qf’ to be constantes of sort State. Moreover, constants ‘q0’ and ‘qf’ are declared to be of sorts InitialState and FinalState, respectively, by means of membership equational axioms (in lines 7 and 8). Finally, rules (in lines 12 to 17) specify the transition rules of the automaton that check if a given word has “aa” or “bb” as subwords.

```
mod AA−BB−SUBWORD is
  ex AUTOMATON .

  ops a b : → Alphabet .
  ops q0 q1 q2 qf : → State .

  mb q0 : InitialState .
  mb qf : FinalState .

  var sigma : Alphabet . var W : Alphabet∗ .

  rl q0 , sigma W ⇒ q0 , W .
  rl q0 , a W ⇒ q1 , W .
  rl q0 , b W ⇒ q2 , W .
  rl q1 , a W ⇒ qf , W .
  rl q2 , b W ⇒ qf , W .
  rl qf , sigma W ⇒ qf , W .
endm
```

Listing 2. AA-BB-SUBWORD system module.

```
============================================
search in AA−BB−SUBWORD : q0,a b b a ⇒∗ epsilon,F:FinalState .

Solution 1 (state 10)
states: 11 rewrites: 17 in 0ms cpu (0ms real) (93406 rewrites/second)
F:FinalState ⟶ qf

No more solutions.
states: 11 rewrites: 17 in 0ms cpu (0ms real) (79812 rewrites/second)
============================================
search in AA−BB−SUBWORD : q0,a b a ⇒∗ epsilon,F:FinalState .

No solution.
states: 7 rewrites: 10 in 0ms cpu (0ms real) (196078 rewrites/second)
```

Listing 3. Word acceptance by an automaton as search.

Now, given a term of sort Configuration representing the initial configuration of a given automaton, acceptance can be implemented by searching for a term of sort Configuration that contains a FinalState and the empty word epsilon. Listing 3 exemplifies that word “a b b a” is accepted by the automaton whose computations are specified in module AA-BB-SUBWORD since the final state ‘qf’ is reached with the empty word. (All symbols were read.) Since there is no solution for a search starting from configuration ‘q0, a b a’ it means that the word “a b a” is not accepted by the automaton.

An example of metaprogramming in Maude. Maude modules can be treated as terms in META-LEVEL, a predefined Maude module that represents an universal theory of meta-represented modules. The module META-LEVEL defines many data structures to manipulate terms and modules at the meta-level. The so-called descent functions in such data structures allow for performing different meta-level computations.

Listing 4 illustrates the same search done at object level (as opposed to meta-level) in Listing 3 for the initial configuration ‘q0 , a b b a’ calling metaSearch

```
mod META-LEVEL-EXAMPLE is
  pr META-LEVEL . pr AA-BB-SUBWORD .
endm
======================================
reduce in META-LEVEL-EXAMPLE :
metaSearch(upModule('AA-BB-SUBWORD, false), upTerm(q0,a b b a),
  upTerm(epsilon,F:FinalState), nil, '*, 4, 0) .
rewrites: 19 in 0ms cpu (0ms real) (34608 rewrites/second)
result ResultTriple:
{'_`,_['epsilon.Alphabet*,'qf.FinalState],'Configuration,
  'F:FinalState <- 'qf.FinalState}
======================================
reduce in META-LEVEL-EXAMPLE :
downTerm(getTerm(metaSearch(upModule('AA-BB-SUBWORD, false),
  upTerm(q0,a b b a), upTerm(epsilon,F:FinalState), nil, '*, 4, 0)),
    error:[Configuration]) .
rewrites: 22 in 0ms cpu (0ms real) (194690 rewrites/second)
result Configuration: epsilon,qf
```

Listing 4. Word acceptance by an automaton as metaSearch.

in the context of a module that includes modules META-LEVEL and AA-BB-SUBWORD. Function metaSearch has a number of parameters: (i) the meta-module where the meta-search will be performed, (ii) the meta-level representation of the initial state of the search, (iii) the meta-level representation of a pattern denoting the states to be reached, (iv) a condition for the meta-seach (which in Listing 4 is empty), (v) an identifier denoting the rewriting relation to be used ('* is used in Listing 4 denoting zero or more rewrites), (vi) a bound for the search, denoting the maximum depth of the search and (vii) the solution number. The result of metaSearch is a term of sort ResultTriple with a meta-term denoting a reachable state, its meta-represented type and a set of substitutions with respect to the pattern in parameter (iii).

Functions upModule and upTerm, also used in Listing 4 are also meta-functions. Not surprisingly, they produce the meta-level representations of a given module and a given term, respectively. The output of metaSearch can be brought to object level using functions getTerm and downTerm. The former projects the first component out of a ResultTriple (the output of metaSearch) and downTerm produces the object level representation of a meta-term or its second argument when it fails to produce the object level representation of the first argument.

4 Associating DL models to rewrite theories

In this Section, we discuss how to associate a DL model, according to Definition 3, to a rewrite theory. As a PDL model $\mathcal{M} = \langle W, R_\pi, \mathbf{V}\rangle$ is a Kripke structure, we may apply the mapping from Kripke structures to rewrite theories defined in [3,4], a technique to represent Kripke structures in the Maude language.

The semantics of PDL (Definition 4) is defined by the satisfaction relation $\mathcal{M}, w \Vdash \varphi$ for $w \in \mathcal{M}$. A rewrite theory is a tuple $\mathcal{R} = \langle (\Sigma, E), R\rangle$ where (Σ, E)

is an equational theory such that Σ is a set of functional symbols and E is a set of equations, and R is a set of rewriting rules. To associate $\mathcal{M}$ to $\mathcal{R}$ we need to specify the kind for states of $\mathcal{M}$ (i.e. W) in Σ and Φ (i.e. the set of atomic propositions — $\mathbf{V}$ as state predicates).

For a module `M` specifying the behavior of a system, the state predicates are defined in an extension of `M` that equationally defines the satisfaction predicate. A sort in `M` must be declared as a subsort of sort State. The syntax of the state predicates is defined by means of operators of sort Prop and the semantics is defined by means of a set of equations that specify for which states a given state predicate evaluates to `true`.

Let k be the kind of states and Π the set of state predicates defined as equations D, an extension of `M`. The definition of the set of atomic propositions in $\mathcal{R}$ given Π is

$$\Phi_{\Pi} = \{\theta(p) \mid p \in \Pi, \theta \text{ ground substitution}\},$$

where $\theta(p)$ is the simplified notation for $\theta(p(x_1, \ldots, x_n))$. This defines a labelling function L_{Π} on the set of states $T_{\Sigma/E,k}$, assigning to each $[t] \in T_{\Sigma/E,k}$ the set of atomic propositions

$$L_{\Pi}([t]) = \{\theta(p) \in \Phi_{\Pi} \mid (E \cup D) \vdash (\forall\emptyset)t \models \theta(p) = \texttt{true}\}.$$

The Kripke structure $\mathcal{K}(\mathcal{R}, k)_{\Pi} = \langle T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R}})^{\bullet}, L_{\Pi}\rangle$ is composed, where $(\rightarrow^1_{\mathcal{R}})^{\bullet}$ denotes the total relation extending the one-step $\mathcal{R}$-rewriting relation $\rightarrow^1_{\mathcal{R}}$ among states of kind k ($[t] \rightarrow^1_{\mathcal{R}} [t']$ holds iff there are $u \in [t]$, $u' \in [t']$ such that u' is the result of applying one of the rules in R to u at some position).

Thus, to verify if a PDL model $\mathcal{M}$ satisfies the formula φ in a state t, it is necessary to implement a procedure to verify the relation

$$\mathcal{K}(\mathcal{R}, k)_{\Pi}, [t] \Vdash \varphi.$$

Section 5 describes Petri-PDL, an extension of PDL to specify and reason about Petri nets. Section 6 discusses a Maude implementation of a bounded model checker for Petri-PDL.

5 Petri-PDL

This section recounts the basic definitions of Petri-PDL, including the restrictions over the Petri net model [9]. In Petri-PDL, the language of Petri nets is restricted to only three types of transition that may be composed. Section 5.1 defines the syntax and semantics of Petri-PDL and Section 5.2 illustrates a specification in the Petri-PDL logic.

5.1 Petri-PDL syntax and semantics

The language of Petri-PDL consists of

Propositional symbols: $p, q \ldots$, where Φ is the set of all propositional symbols

Place names: e.g.: $a, b, c, d \dots$

Petri net composition symbol: $\odot$

Multiset of names: $S = \{\epsilon, s_1, s_2, \dots\}$, where ϵ is the empty marking. The notation $s \prec s'$ denotes that all names occurring in s also occur in s', regardless its order.

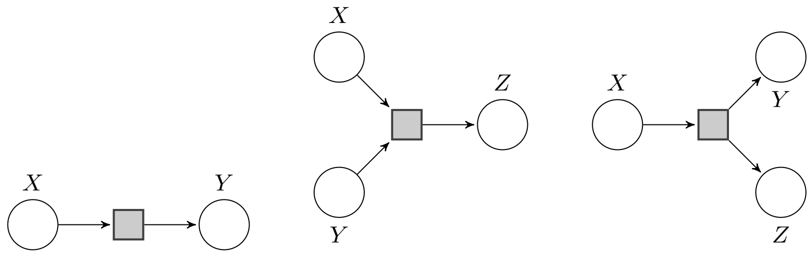
Definition 5 (Petri-PDL program). *Let π denote a Petri net program and s a multiset of names (the marking of π). The transitions may be from three types, $T_1 : xt_1y$, $T_2 : xyt_2z$ and $T_3 : xt_3yz$, each transition has a unique type.*

Basic programs: $\pi_b ::= at_1b \mid abt_2c \mid at_3bc$ *where t_i is of type $T_i, i = 1, 2, 3$*

Composed Petri net programs: $\pi ::= \pi_b \mid \pi \odot \pi$

Marked Petri net programs: $\pi_m ::= s, \pi$

These basic Petri nets in Definition 5 are illustrated in Fig. 3.



(a) *Type* 1 : t_1 (b) *Type* 2 : t_2 (c) *Type* 3 : t_3

Fig. 3. Basic Petri nets

Definition 6 (Petri-PDL formula). *A formula is defined as*

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle s, \pi \rangle \varphi.$$

The standard abbreviations $\bot \equiv \neg\top$, $\varphi \vee \phi \equiv \neg(\neg\varphi \wedge \neg\phi)$, $\varphi \to \phi \equiv \neg(\varphi \wedge \neg\phi)$ and $[s, \pi]\varphi \equiv \neg\langle s, \pi \rangle \neg\varphi$ are valid.

Definition 7 (Firing relation). *The* firing *relation $f : S \times \pi \to S$ is defined as follows.*

- $f(s, at_1b) = \begin{cases} s_1bs_2 \text{ } \textit{if } s = s_1as_2 \\ \epsilon \text{ } \textit{if } a \not\prec s \end{cases}$
- $f(s, abt_2c) = \begin{cases} s_1cs_2s_3 \text{ } \textit{if } s = s_1as_2bs_3 \text{ } \textit{or } s = s_1bs_2as_3 \\ \epsilon \text{ } \textit{if } a \not\prec s \text{ } \textit{or } b \not\prec s \end{cases}$
- $f(s, at_3bc) = \begin{cases} s_1s_2bc \text{ } \textit{if } s = s_1as_2 \\ \epsilon \text{ } \textit{if } a \not\prec s \end{cases}$
- $f(\epsilon, \pi) = \epsilon$, *for all Petri nets programs π.*

Definition 8 (Petri-PDL frame). *A frame for Petri-PDL is a tuple $\mathcal{F} = \langle W, w_0, R_\pi, M\rangle$, where*

- *W is a non-empty set of states;*
- *w_0 is the initial state;*
- *$M: W \to S$;*
- *R_π is a binary relation over W, for each basic program π, satisfying the following condition. Let $s = M(w)$*
  - *if $f(s,\pi) \neq \epsilon$, $wR_\pi v$ iff $f(s,\pi) \prec M(v)$*
  - *if $f(s,\pi) = \epsilon$, $wR_\pi v$ iff $w = v$*

Definition 9 (Behaviour of R over composed Petri Net programs). *The behaviour of the relation R_π is inductively defined, for each Petri Net program $\pi = \pi_1 \odot \pi_2 \odot \ldots \odot \pi_n$, as*

$$R_\pi = \{(w,v) \mid \text{ for some } \pi_i, \exists u \text{ such that } s_i \prec M(u) \text{ and } wR_{\pi_i}u \text{ and } uR_\pi v\}$$

where $s_i = f(s,\pi_i)$, for all $1 \leq i \leq n$ and $s = M(w)$.

Definition 10 (Petri-PDL model). *A* model *for Petri-PDL is a pair $\mathcal{M} = \langle \mathcal{F}, \mathbf{V}\rangle$, where $\mathcal{F}$ is a Petri-PDL frame and $\mathbf{V}$ is a valuation function $\mathbf{V}: W \to 2^{\Phi}$.*

Definition 11 (Petri-PDL satisfaction notion). *Let $\mathcal{M} = \langle \mathcal{F}, \mathbf{V}\rangle$ be a model. The notion of* satisfaction *of a formula φ in a model $\mathcal{M}$ at a state w, notation $\mathcal{M}, w \Vdash \varphi$, can be inductively defined as follows:*

- *$\mathcal{M},w \Vdash p$ iff $w \in \mathbf{V}(p)$;*
- *$\mathcal{M},w \Vdash \top$ always;*
- *$\mathcal{M},w \Vdash \neg\varphi$ iff $\mathcal{M},w \nVdash \varphi$;*
- *$\mathcal{M},w \Vdash \varphi_1 \wedge \varphi_2$ iff $\mathcal{M},w \Vdash \varphi_1$ and $\mathcal{M},w \Vdash \varphi_2$;*
- *$\mathcal{M},w \Vdash \langle s,\eta\rangle\varphi$ iff there exists $v \in W$, $wR_\eta v$, $s \prec M(w)$ and $\mathcal{M},v \Vdash \varphi$.*

If $\mathcal{M},v \Vdash A$ for every state v then A is *valid in the model* $\mathcal{M}$, denoted $\mathcal{M} \Vdash A$. And if A is valid in all $\mathcal{M}$ then A is *valid*, denoted $\Vdash A$. Petri-PDL is proved to be sound, complete and decidable regarding its semantics [8,9].

5.2 A Petri-PDL example

As a Petri-PDL example, already discussed in [9], take the Petri net of Fig. 4. It models a "Rock-Paper-Scissors". When a token is placed into the "input" (i.e. the place "C" has a token), a transition may fire and the place "G$_1$" and "G$_2$" will have a token. Now each player can select one of the options. If the user wins, a token will be placed at "W$_1$; if he loses, at the place "W$_2$" or at "D" if there is a draw match.

Modelling this situation in a Petri-PDL program we have $\pi = ct_3g_1g_2 \odot g_1t_1r_1 \odot g_1t_1s_1 \odot g_1t_1p_1 \odot g_2t_1r_2 \odot g_2t_1s_2 \odot g_2t_1p_2 \odot r_1s_2t_2w_1 \odot r_1p_2t_2w_2 \odot r_1r_2t_2d \odot s_1r_2t_2w_2 \odot s_1s_2t_2d \odot s_1p_2t_2w_1 \odot p_1r_2t_2w_1 \odot p_1s_2t_2w_2 \odot p_1p_2t_2d$. Let $p, q \in \Phi$, two propositional symbols where p holds iff G_1 wins (i.e. there is a token in W_1) and q holds iff G_2 wins (i.e. there is a token in W_2). To verify if, after the game, there will be always a winner is equivalent to verify if the formula $\neg\langle(c),\pi\rangle\neg(p \vee q)$ holds in this model.

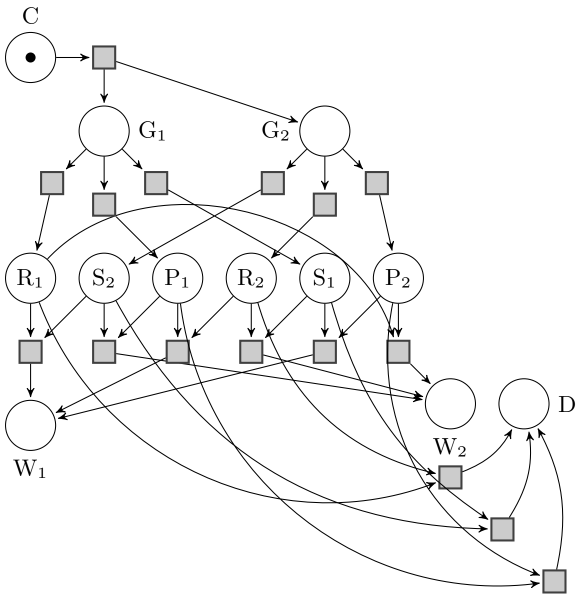


Fig. 4. Petri Net for "Rock-Paper-Sicissors" game

6 A prototype model checker for Petri-PDL in Maude

We have prototyped a bounded model checker for Petri-PDL by extending the mapping from Kripke structures to Rewriting Logic, recalled in Section 4, to Petri-PDL. It is available for download from http://www.ic.uff.br/∼cbraga/petri-pdl-sbmf15.maude.

In module PETRI-PDL (see Listing 5) we define the syntax of Petri-PDL programs (see Definition 5). Markings of a Petri net are represented by terms of sort Places. Program composition ("$\odot$" in Petri-PDL) is denoted by operation +. The firing relation (see Definition 7) is specified by the rules of module PETRI-PDL. They describe how Petri nets (terms of sort Net, composed by places and a program) evolve by the non-deterministic evaluation of basic programs.

The module PETRI-PDL-MODEL-CHECKER (see Listing 6) includes modules PETRI-PDL and META-LEVEL (that implements Maude's metaprogramming API). It declares, among other auxiliary data structures, functions

$$\begin{aligned}&\text{modelCheck} : \text{Formula Nat PlacesListSet} \to \text{PPDLModel},\\&\text{ppdlNStepRewTrace} : \text{NetNat} \to \text{PlacesListSet}\end{aligned}$$

and predicate

$$\text{holdsIn} : \text{Formula Places} \ \to \text{Bool}.$$

Function modelCheck is equationally defined in lines 32 to 36 of Listing 6. It is responsible for the implementation of the satisfaction predicate in Defini-

```
mod PETRI−PDL is
  sort Place Places BasicProg Prog Net .
  subsort Place < Places .
  subsort BasicProg < Prog .
  ops epsilon a b c d e l m x y c g1 g2 s1 s2 r1 r2 p1 p2 w1 w2 : → Place .
  op __ : Places Places → Places [prec 20 assoc comm id: epsilon] .
  op _t1_ : Place Place → BasicProg [prec 30] .
  op __t2_ : Place Place Place → BasicProg [prec 30] .
  op _t3__ : Place Place Place → BasicProg [prec 30] .
  op _+_ : Prog Prog → Prog [assoc comm prec 40] .
  op _,_ : Places Prog → Net .
  op getPlaces : Net → Places .

  vars A B C : Place . var W : Places . var P : Prog .

  eq getPlaces(W, P) = W .

  rl [t1] : A W , A t1 B ⇒ B W , A t1 B .
  rl [t2] : A B W , A B t2 C ⇒ C W , A B t2 C .
  rl [t3] : A W , A t3 B C ⇒ B C W , A t3 B C .

  rl [t1] : A W , A t1 B + P ⇒ B W , A t1 B + P .
  rl [t2] : A B W , A B t2 C + P ⇒ C W , A B t2 C + P .
  rl [t3] : A W , A t3 B C + P ⇒ B C W , A t3 B C + P .
endm
```

Listing 5. Excerpt of PETRI-PDL Maude module.

tion 11. It is inductively defined on the syntax of Petri-PDL language. Function ppdlNStepRewTrace is equationally defined in lines 42 to 48, considering its auxiliary function. It implements relation R_π in Definitions 8 and 9. Predicate holdsIn is equationally defined in line 39. Intuitively, it produces a set of traces (terms of sort PlacesListSet) by, roughly, stepping ppdlOneStepRewTrace n times, for each term in depth $n-1$, where n is the bound in the call to function modelCheck. Function ppdlOneStepRewTrace essentially calls function ppdlSearch that, by invoking Maude's metaSearch function, returns the set of reachable states from an initial one on a given depth. Finally, predicate holdsIn, equationally defined in line 39, verifies if a given formula holds in a marking (a term of sort Places). It invokes function metaReduce, from Maude's reflexive API, on operation valuation, on a given marking, equationally defined in module VALUATION. Such a reduction produces a set of atomic propositions that function applyValuation uses to check if a given formula holds or not on a given marking.

In summary, given a PDL model $\mathcal{M} = \langle W, R_\pi, \mathbf{V}\rangle$, sort Places represents set W. Relation R_π is implemented by function ppdlNStepRew, which encapsulates a call to metaSeach function from Maude meta-level API. Function modelCheck is responsible for the implementation of Petri-PDL' satisfaction predicate. The valuation function $\mathbf{V}$ is defined in module VALUATION that provides equations for operation valuation defining which atomic propositions hold on Places.

Listing 7 gives the valuation function for the Petri-PDL example in Section 5.2 where w1, w2 and d denote the situations where the first player wins, the second player wins, and when there is a draw, respectively. The valuation

```
mod PETRI−PDL−MODEL−CHECKER is inc PETRI−PDL . inc META−LEVEL .
  sort PlacesList . subsort Places < PlacesList .
  op mt−placeslist : → PlacesList .
  op _−>_ : PlacesList PlacesList → PlacesList [assoc id: mt−placeslist] .

  sort PlacesListSet . subsort PlacesList < PlacesListSet .
  op mt−placeslistset : → PlacesListSet .
  op _;;_ : PlacesListSet PlacesListSet → PlacesListSet [assoc comm id: mt−placeslistset] .

  sort PPDLModel .
  op ppdlModel : Bool PlacesList → PPDLModel .
  op modelCheck : Formula Nat PlacesListSet → PPDLModel .
  op holdsInTrace : Formula PlacesListSet → PPDLModel .

  sort PDLFormula . subsort PDLFormula < Formula .
  op <_,_>_ : Places Prog Formula → PDLFormula .
  op valuation : Places → FormulaSet .
  op holdsIn : PDLFormula Places → Bool .

  op ppdlNStepRewTrace : Net NzNat → PlacesListSet .
  op ppdlNStepRewTrace−aux : Prog NzNat PlacesListSet → PlacesListSet .
  op ppdlOneStepRewTraces : Net PlacesList → PlacesListSet .
  op ppdlOneStepRewTraces−aux : Net Nat PlacesList PlacesListSet → PlacesListSet .
  op ppdlSearch : Net Nat Nat → [Places] .
  op no−solution : → [Places] . op error : → [Net] .

  vars BOUND SOL N : Nat . var W W1 W2 : Places . var P : Prog . var E : Net .
  vars PDLF PDLF1 PDLF2 : PDLFormula . vars A F F1 F2 : Formula . var Q : Qid . var FS :
        FormulaSet .
  var PL PL1 PL2 : PlacesList . var PLS PLS1 PLS2 : PlacesListSet . var B : Bool . var
        PPDLM : PPDLModel .

  −−− PPDL model checking
  eq modelCheck(¬ PDLF, N, PLS) = not modelCheck(PDLF, N, PLS) .
  eq modelCheck(PDLF1 \/ PDLF2, N, PLS) = if isTrue?(modelCheck(PDLF1, N, PLS)) then
        modelCheck(PDLF1, N, PLS) else modelCheck(PDLF2, N, PLS) fi .
  eq modelCheck(PDLF1 /\ PDLF2, N, PLS) = if isTrue?(modelCheck(PDLF1, N, PLS)) then
        modelCheck(PDLF2, N, PLS) else modelCheck(PDLF1, N, PLS) fi .
  eq modelCheck(< W, P > F, N, PLS) = include(PLS, modelCheck(F, sd(N,1),
        ppdlNStepRewTrace((W,P), N))) .
  ceq modelCheck(F, N, PLS) = holdsInTrace(F, PLS) if PLS =/= mt−placeslistset [owise] .

  −−− holdsIn: Checks if a given propositional formula holds in a marking.
    eq holdsIn(F, W) = applyValuation(F, downTerm(getTerm(metaReduce(upModule('
          VALUATION, false), 'valuation[upTerm(W)])), formula−set−error)) .

  −−− ppdlNStepRewTrace: Implements R_n relation, that is, produces all the traces after n
        steps from a given Net.
  eq ppdlNStepRewTrace((W,P), 0) = W .
  eq ppdlNStepRewTrace((W,P), N) = ppdlNStepRewTrace−aux(P, N, W) .

  eq ppdlNStepRewTrace−aux(P, 0, PLS) = PLS .
  eq ppdlNStepRewTrace−aux(P, N, mt−placeslistset) = mt−placeslistset .
  ceq ppdlNStepRewTrace−aux(P, N, ((PL −> W) ;; PLS)) =
    ppdlNStepRewTrace−aux(P, sd(N, 1), ppdlOneStepRewTraces((W,P), PL) ;;
          ppdlNStepRewTrace−aux(P, 1, PLS)) if N > 0 .

  −−− ppdlSearch: Searches for the SOLth soultion, in BOUND depth, from a given Net.
    eq ppdlSearch((W, P), BOUND, SOL) =
       if downTerm(getTerm(metaSearch(upModule('PETRI−PDL, false), upTerm((W, P)), 'N:
             Net, nil, '+ , BOUND, SOL)), error) == error
       then no−solution
       else getPlaces(downTerm(getTerm(metaSearch(upModule('PETRI−PDL, false), upTerm((
             W, P)), 'N:Net, nil, '+ , BOUND, SOL)), error))
       fi .
  . . .
endm
```

Listing 6. Excerpt of PDL-MODEL-CHECKER Maude module.

```
mod VALUATION is
  inc PETRI−PDL−MODEL−CHECKER .
  ops c g1 g2 s1 s2 r1 r2 p1 p2 w1 w2 d : → Place .
  ops p q : → Formula .
  eq valuation(w1) = p . eq valuation(w2) = q . eq valuation(d) = ((¬ p) (¬ q)) .
endm
```

Listing 7. Excerpt of VALUATION Maude module.

```
reduce in VALUATION : modelCheck(¬ < c,(g1 t1 r1 + g1 t1 p1 + g1 t1 s1 + g2 t1 r2 + g2 t1
      p2 + g2 t1 s2 + (((((((
    s1 s2 t2 d + s1 p2 t2 w1) + s1 r2 t2 w2) + p1 s2 t2 w2) + p1 p2 t2 d) + p1 r2 t2 w1) + r1 s2
          t2 w1) + r1 p2
    t2 w2) + r1 r2 t2 d) + c t3 g1 g2 > (¬ (p \/ q)), 4, mt−placeslistset) .
rewrites: 1139 in 24ms cpu (25ms real) (45942 rewrites/second)
result PPDLModel: ppdlModel(false, c −> g1 g2 −> g1 s2 −> s1 s2 −> d)
```

Listing 8. Model checking the Rock-paper-scissors example.

function formalizes that atomic propositions p and q are true in w1 and 2, respectively and that neither is true when there is a draw.

Finally, Listing 8 illustrates the use of Petri-PDL bounded model checker. It provides a counter-example when we ask if it is always true that there is a winner in the example of Section 5.2. A trace reaching the draw state is then returned.

7 Related work

There is a large body of literature regarding Petri nets. Trace Theory [10] presents a highly expressive algebra without restrictions over the Petri net model but without decidability and completeness.

The use of PDL as a query language for Petri nets was already proposed [14], but PDL does not allows for reasoning directly on Petri nets. The net must be translated to the usual regular program of Petri nets, but as there is not a bijection between Petri nets and PDL programs: information is lost. There is an implementation of a model checker for PDL with graph systems [7]. Their approach relies on graph transformations with second order μ-calculus.

In [13] the authors discuss many connections between Rewriting Logic and Petri nets, but in a categorical approach, unrelated to the proof-theoretical PDL-based aspects that this work builds on.

8 Conclusion

We have discussed a prototype implementation of a model checker for Petri nets based on Propositional Dynamic Logic. Our model checker is formally designed by the application of a general mapping from Kripke structures to rewrite theories [4] to Petri-PDL, an extension of PDL for Petri nets [9].

Future work, for Petri-PDL model checker, includes a more efficient implementation that takes further advantage of the Rewriting Logic calculus. Our long term goal is to evolve this tool into a tool set for reasoning in Dynamic Logics.

References

1. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theor. Comput. Sci.*, 236(1-2):35–132, Apr. 2000.
2. M. Bourcerie, F. Bousseau, and F. Guegnard. Petri Nets for production systems: Teaching and research in europe. In *Global Cooperation in Engineering Education: Innovative Technologies, Studies and Professional Development - International Conference Proceedings*, pages 85–89, 2008.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
4. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker and its Implementation. In *Proceedings of the 10th international conference on Model checking*, pages 230–234. Springer-Verlag, 2003.
5. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.
6. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing Series. MIT Press, 2000.
7. A. L. Lafuente and A. Vandin. Towards a Maude Tool for Model Checking Temporal Graph Properties. In *Proceedings of theTenth International Workshop onGraph Transformation andVisual Modeling Techniques*, volume 41, pages 1–14, 2011.
8. B. Lopes, M. Benevides, and E. H. Haeusler. Extending Propositional Dynamic Logic for Petri Nets. *Electronic Notes in Theoretical Computer Science*, 305(11):67–83, 2014.
9. B. Lopes, M. Benevides, and E. H. Haeusler. Propositional dynamic logic for Petri nets. *Logic Journal of the IGPL*, 22(5), 2014.
10. A. Mazurkiewicz. Trace theory. In W. Brauer., W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 278–324. Springer, 1987.
11. C. Nalon, B. Lopes, E. H. Haeusler, and G. Dowek. A calculus for automatic verification of Petri Nets based on Resolution and Dynamic Logics. *Electronic Notes in Theoretical Computer Science*, 312:125–141, 2015.
12. C. A. Petri. Fundamentals of a theory of asynchronous information flow. *Communications of the ACM*, 5(6):319–319, 1962.
13. M.-O. Stehr, J. Meseguer, and P. C. Ölveczky. Rewriting logic as a unifying framework for petri nets. In *Unifying Petri Nets, Advances in Petri Nets*, pages 250–303, London, UK, UK, 2001. Springer-Verlag.
14. H. Tuominen. Elementary net systems and Dynamic Logic. In G. Rozenberg, editor, *Advances in Petri Nets 1989*, Lecture Notes in Computer Science, pages 453–466. Springer Berlin Heidelberg, 1990.
15. R. Zurawski and M. C. Zhou. Petri Nets and industrial applications – a tutorial. *IEEE Transactions on Industrial Electronics*, 41(6):567–583, 1994.

Evaluating the Assignment of Behavioral Goals to Coalitions of Agents

Christophe Chareton[1], Julien Brunel[2], David Chemouil[2]

[1] École Polytechnique de Montréal, Montreal (Quebec), Canada
[2] Onera/DTIM, Toulouse, France

Abstract. We present a formal framework for solving what we call the "assignment problem": given a set of behavioral goals for a system and a set of agents described by their capabilities to make the system evolve, the problem is to find a "good" assignment of goals to (coalitions of) agents. To do so, we define Kore, a core modelling framework as well as its semantics in terms of a strategy logic called USL. In Kore, agents are defined by their capabilities, which are pre- and post-conditions on the system variables, and goals are defined in terms of temporal logic formulas. Then, an assignment associates each goal with the coalition of agents that is responsible for its satisfaction. Our problem consists in defining and checking the *correctness* of this assignment. We define different criteria for modelling and formalizing this notion of correctness. They reduce to the satisfaction of USL formulas in a structure derived from the capabilities of agents. Thus, we end up with a procedure for deciding the correctness of the assignment. We illustrate our approach using a toy example featuring exchanges of resources between a provider and two clients.

1 Introduction

The question of assigning behavioral goals to coalitions of agents (*i.e.* sets of active entities), the capabilities of whom are known, is a fundamental and recurring problem in various areas of software and systems engineering. We call it the *assignment problem*. In this paper, we propose a formalization of this problem and then describe various criteria to assess an assignment formally.

First, let us illustrate various situations where this problem arises; this will not only demonstrate the ubiquity of this problem but also enable us to delineate the most salient aspects that ought to be addressed in the formalization.

In the field of Requirements Engineering (RE), for instance, several modeling languages have been proposed that each partly feature the concepts just mentioned. Thus, Kaos [Let02, LVL02b, vL09, vL03], a so-called *goal-oriented* modeling language, features *behavioral* goals that may be formalized using Linear Temporal Logic (LTL). This allows us to assert and check the correctness of both refinement between goals and of realization of goals by operations. On the other hand, *agent-oriented* modeling languages, such as Tropos [BPG$^+$04] and i^* [Yu09, Yu96], also focus on the agents that will realize these goals. A formal extension of i^*, featuring *commitments* and *protocols* [CDGM10b, CDGM10a, CS09, MS06], aims at checking the capabilities of agents to ensure the satisfaction of the goals. There, goals are described using propositional logic and agents are described along with their capabilities to ensure the satisfaction of propositional formulas. Thus, the need for a treatment of the assignment problem has been identified in RE but, to the best of our knowledge, no proposition has been made until now to address it for behavioral goals, in a formalized framework.

In *Systems-of-Systems Engineering* (SoSE) [Mai98], several independent systems, made up of subsystems (agents in our parlance) interact altogether to achieve a global mission. Consider one of these systems and its set of agents A. Then, to investigate whether the agents in A are able to ensure a given goal in the global system, one must take into account the side effects of the actions performed by the other agents (from other systems) pursuing different goals.

Finally, in Component-Based Software Engineering (CBSE) [Szy02], individual components (agents, for us) may be assembled into composite subsystems in order to fulfill requirements specifications. The capabilities of these agents are given as contracts [BJPW99]. Then knowing whether the resulting architecture indeed satisfies its specification is of major importance. Besides, identifying unsatisfied specifications can provide guidance to the engineer for adding new components. Identifying unexpected side effects (good or bad) between components is also very important.

These various examples lead us to propose the following informal characterization of the assignment problem:

Definition 1 (Assignment Problem, Informally). *Given a set of interacting agents, the capabilities of whom are known, given a set of goals, and given an* assignment function *mapping each goal to a coalition (*i.e. *a set) of agents, is every goal assigned to a coalition of agents who are able to ensure its satisfaction (including by benefiting from actions of other coalitions)?*

The objective of this paper is to formalize this definition and to provide a means to solve the assignment problem. In particular, notice that in this definition, what *interaction* is is left ambiguous. One of our contributions is precisely to propose multiple acceptations, each of them inducing a particular case of the problem. We call these cases *correctness criteria* for an assignment. We model the assignment problem and these criteria, and we describe a formal process to check their satisfaction for any instance.

Our approach was originally developed for agent- and goal-oriented RE [CBC11]. In this field, to the best of our knowledge, this provides the first unified formal framework addressing the satisfaction of *behavioral* goals by operation specifications and the capabilities of agents to perform these operations as required.

Checking the capabilities of agents to ensure goals also enables one to distinguish, given a set of available agents and a set of goals, those goals that the available agents *cannot* ensure. Then these goals can support the engineer in identifying *new* agents that should be introduced to fulfill all goals.

Our approach also makes it possible to characterize other sorts of interaction phenomena. Here, we stress the following:

- First, given two coalitions of agents and a goal for each coalition, we highlight *dependencies* between coalitions w.r.t. the satisfaction of their respective goals.

- Second, in an SoS for instance, we can check whether, while pursuing their own goals, agents in a system S_1 necessarily entail, as a *side effect* on the global system, that agents in a system S_2 can also ensure their own goals.

The remainder of this article is organized as follows: in Sect. 2 we introduce a minimal modeling framework, called Kore, to allow a proper representation of the situations that we wish to address. In practice, this framework can be seen as a subset of modeling languages such as Kaos or SysML. In the same section, we also introduce a minimal example which will be used in the following sections to illustrate our approach. In Sect. 3, we give a description of the assignment problem in the framework introduced by Sect. 2. To do so, we analyze various modalities of interactions between agents and we devise corresponding correctness criteria for the assignment. This way, the assignment problem is reduced to the problem of satisfaction of the evaluation criteria by an instance of Kore. Then, this problem is itself reduced to a model-checking problem for a multi-agent logic called USL in Sect. 4. Related work is discussed in Sect. 5.

2 A Modeling Framework for the Assignment Problem

As sketched before, a Kore model is described using a set of goals to be realized, a description of the context given by a number of *context properties*, a set of agents (considered with their *capabilities* to act on the system) and an *assignment* of the goals to (coalitions of) agents. Goals and context properties are temporal properties, so we briefly introduce in Sect. 2.1 the logic that we use to formalize them.

Let us first introduce a running example that will be used throughout this paper to illustrate our approach. In this example, we consider a resource, provided by a *provider*. Then two clients A and B have different needs w.r.t. this resource. As we will proceed through this article, we will envision three variations of this example.

Example 1 (CP_1). In the first version (CP_1), the provider can provide up to 15 units of the resource per time unit. It can also decide to which clients the resources are affected. Concretely, at each time unit it provides the clients up to 15 new units as a whole, distributed in variables new_A for client A and new_B for client B. Each client is able to receive up to 15 units of the resource at a time.

2.1 *LTL*$_{\text{Kore}}$

In our setting, goals are behavioral: therefore we formalize them as well as context properties in a version of Linear Temporal Logic (LTL, [MP95]). Following [Let02,LVL02b, vL09, vL03], we describe an action (of an agent) as the modification of the values of variables describing the state of the sytem. Therefore, in our version of LTL(called *LTL*$_{\text{Kore}}$), atomic propositions are comparisons of integer variables and constants.

Definition 2 (Cond). *Let X be a set of variables, the set of propositions Cond over X (written Cond (X)) is given by the following grammar:*

$$\varphi ::= x \sim n \mid x - y \sim n \mid x + y \sim n \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi$$

where $x, y \in X$, n is a constant in $\mathbb{Z}$, and $\sim \in \{<, >, =, \leqslant, \geqslant\}$.

Definition 3 (*LTL*$_{\text{KORE}}$). *Let X be a set of variables, the logic* $LTL_{\text{Kore}}(X)$ *is the usual Linear Temporal Logic where atoms are taken from Cond*(X)*. It is generated by the following grammar:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi \quad \textit{where } p \in \textit{Cond}(X).$$

Because LTL is standard knowledge in formal approaches and due to space constraints, we do not detail the formal semantics of LTL and refer the interested reader to [MP95]. Basically,an $LTL_{\text{Kore}}(X)$ formula is interpreted over discrete traces of assignments (instants) for variables in X. A formula $\mathbf{X}\varphi$ is true in one of these traces, at a given instant, iff φ is true in the same trace at the next instant. A formula $\Box\varphi$ is true in a trace iff φ is true at every instant of this trace.

Example 2 (CP_1 *(cont.)).* Version CP_1 of the *client-provider* example makes use of the following variables for every $c \in \{A, B\}$:

- a variable res_c denotes the amount of resources available for client c at any time $t \geqslant 1$. This amount is the sum of:
  - the part remaining from the resource in res_c at time $t-1$, denoted old_c
  - the resources added by *provider* in res_c at transition from time $t-1$ to t: new_c
- at any time $t \geqslant 2$, obt_c denotes the amount of resources obtained by c from the amount in store res_c at time $t-1$.

We also write X_{CP} for the set of variables $\bigcup_{c\in\{A,B\}}\{res_c, old_c, new_c, obt_c\}$.

Now that LTL_{Kore} has been introduced, let us delve into the concepts necessary to consider the assignment problem as we informally defined it in Sect. 1.

2.2 Goals

In Kore, *goals* are statement describing the behavior expected from the system. They are formalized in LTL_{Kore}.

Example 3 (CP_1 *(cont.)).* In CP_1, every client wishes to get a certain amount of the resource: A (resp. B) wants to get at least 6 (resp. 12) units of the resource per unit of time $t \geqslant 2$. Their respective goals g_A and g_B are formalized as follows[3]:

$$[\![g_A]\!] \triangleq \mathbf{XX}(\Box(obt_A \geqslant 6)) \qquad [\![g_B]\!] \triangleq \mathbf{XX}(\Box(obt_B \geqslant 12))$$

2.3 Context Properties

Context properties are statements describing the system as it is (as opposed, for instance, to expected properties). In this paper:

- They may be used to specify the set of states of the system, in which case we call them *static*.

[3] We use double brackets $[\![\cdot]\!]$ to denote the formalization of an informal property.

- They may also concern the initial state of the system. In this case, they are called *initial* properties. They can be formalized using Cond only.

Example 4 (CP_1 (cont.)). In CP_1 we consider the following context properties:

- Static properties:

  Res_c For any client c, the set of resources in res_c is the union of old_c and new_c. Formally, at any time $t \geqslant 1$, the resources available are the sum of the previous ones (remaining from the value of res_c at time $t-1$) and the new ones, gotten from *provider*:

  $$[\![Res_c]\!] \triangleq \Box(res_c = old_c + new_c)$$

  Obt_c At any time $t \geqslant 1$, when a client c takes some resources from the set res_c, then at time $t+1$, the set old_c is the set of resources remaining from res_c at time t: at any time $t \geqslant 1$ the amount of resources in old_c is the amount in res_c minus the amount obtained by c at transition from time $t-1$ to t:

  $$[\![Obt_{c,k}]\!] \triangleq \Box(res_c = k \rightarrow \mathbf{X}(old_c = k - obt_c))$$

  $Posi_x$ All the variables in the example stand for the cardinality of some set of resources. Therefore they always have a non-negative value. For each $x \in X_{\mathrm{CP}}$:

  $$[\![Posi_x]\!] \triangleq \Box x \geqslant 0$$

- Initial property $Init_x$: in the initial state of the system, there is none of the resource anywhere. Thus, every variable is initialized to 0. For each $x \in X_{\mathrm{CP}}$:

  $$[\![Init_x]\!] \triangleq x = 0$$

2.4 Agents and Capabilities

Agents are the active entities of the system, likely to ensure or prevent the satisfaction of goals. They are described along with their *capabilities*. To define the latter, we need to consider a restriction of Cond to a fragment which characterizes finite intervals (*windows*) only:

Definition 4 (Window conditions). *Let X be a set of variables, the language of window conditions, written $Cond^{win}(X)$, is generated by the following grammar:*

$$\varphi ::= a \leqslant x \wedge x \leqslant b \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \qquad \text{(with } x \in X, a \leqslant b \text{ and } a, b \in \mathbb{Z}\text{).}$$

Definition 5 (Capability). *A capability Cap for an agent a is a pair of a* pre-condition *Cap.enabCond in $Cond(X)$ and of a* window *Cap.window, in $Cond^{win}(X)$.*

The meaning of a capability is as follows: in each state where the *enabCond* holds, the corresponding agent can give to the variables appearing in *window* any values satisfying this *window*. Indeed, our modeling considers that an agent able to act upon variables is not necessarily able to give them *any* value at *any* time. The ∧ connective in Def. 4, is in particular used to link bounds on different variables (see Ex. 5 below). The enabling condition defines the conditions under which an agent can use her capability and change the values of some variables, and the window bounds the set of values she can give to these variables. However, in the example we develop in this article, each window is reduced to a singleton.

Example 5 (Capabilities for all variations of client-provider). The capabilities for the agents in *client-provider* are given hereafter:

Caps. $\{provide\}_{k+\ell \leqslant 15}$ for *provider*:	Caps. $\{get_c\}_{0<k\leqslant 15}$ for client $c \in \{A, B\}$:
$enabCond : \top$	$enabCond : res_c \geqslant k$
$window : new_A = k \wedge new_B = \ell$	$window : obt_c = k$

2.5 Assignment

An assignment is simply a function from goals to coalitions (where every coalition is a set of agents). This can model different kinds of situations; for instance:

- Each agent may be pursuing her own goals (*distributed intentionality* [Yu09]). In this case the assignment models the relation between a goal and the agent(s) aiming for it.
- Or there may be a controller or an engineer, able to constrain and schedule every agent in the model, and who is responsible for the realization of the whole set of goals. In this case the assignment is an affectation, by this controller, of the available resources (the agents) to the satisfaction of the goals.

Example 6 (CP_1 (cont.)). Here we follow the second item, then *provider* is committed to the realization of both goals, so the assignment $\mathcal{A}_1$ is defined by: $\mathcal{A}_1(g_A) = \{provider, A\}$ and $\mathcal{A}_1(g_B) = \{provider, B\}$.

3 Evaluation Criteria for Assignment

Now that the different elements of Kore are defined, let us come back to the assignment problem and seek precise criteria modeling the notion of correctness for an assignment. We first give informal definitions for these criteria. Their formalization, which requires first the introduction of the USL framework, is given in Sect. 4.

3.1 Local correctness

The first version of the assignment problem we consider is the question whether each goal is assigned to a coalition able to ensure it, whatever the other agents do. We call this criterion the *local correctness* of the system under consideration. For the *client-provider* example, this is the question whether $\{provider, A\}$ is able to ensure the satisfaction of g_A (whatever B does) and $\{provider, B\}$ is able to ensure the satisfaction of g_B (whatever A does). We write $LC_{\mathcal{A}}(G)$ for the satisfaction of the local correctness of a set of goals G under an assignment $\mathcal{A}$.

3.2 Global correctness

The criterion of local correctness is easy to understand and to check. Nevertheless it is not sufficient when, as is the case of CP_1, one agent is part of several coalitions being assigned different goals. Indeed, *provider* is able to take part separately in both coalitions $\{provider, A\}$ and $\{provider, B\}$. But the local correctness does not say whether *provider* is able to take part in both coalitions *at the same time*. What *provider* has to do with the first coalition might be contradictory with what she has to do with the second coalition. To overcome this issue, we introduce a second correctness criterion, called the *global correctness*. Global correctness is satisfied if there is a *general behaviour b* of all agents s.t. for each goal g, knowing that the coalition of agents assigned to g behaves according to b is enough to ensure g, whatever the other agents do. The notion of such a *general behaviour* is to be defined as a *multi-strategy profile* in a CGS, in Sect. 4. If the assignment $\mathcal{A}$ of a set of goals G is globally correct, then we write $GC_{\mathcal{A}}(G)$.

3.3 Collaboration

The global correctness criterion is sufficient to ensure that each goal is assigned to a capable coalition. Nevertheless, it may require more than what is necessary. Indeed, it requires that each coalition is able to ensure its goal in a completely autonomous way (whatever the agents not in this coalition do). In certain cases, it may be necessary to soften this criterion and to admit that some given coalitions depend on others to ensure their goals. To illustrate this point, let us slightly modify our example and consider its second version, CP_2.

Example 7 (CP_2). It brings the following changes from CP_1:

- *provider* can produce up to 20 units at a time
- in the new assignment $\mathcal{A}_2$, *provider* is assigned a new goal $g_{provider}$, to produce at least 16 units of the resource per time unit. Furthermore, g_A and g_B are respectively assigned to $\{A\}$ and $\{B\}$.

In this second version, *provider* is able, at the same time, to ensure the satisfaction of its goal and to help A and B. By producing, for example, at least 7 units in new_A and 13 units in new_B, it ensures $g_{provider}$ and the global correction of the model reduced to $\{g_A, g_B\}$. In this case we say that the coalition that is assigned to $g_{provider}$ *globally collaborates* to the satisfaction of $\{g_A, g_B\}$, and we write $Coll_{\mathcal{A}_2}(g_{provider}, \{g_A, g_B\})$.

Note that a relation of *local collaboration* could also be defined a similar way.

3.4 Contribution

In the three criteria introduced above, we adopted the point of view of an engineer controlling every agent in the system. Thus, we considered our model as a closed system and only asked the possibility for a unique decision-maker to specify the agents so that all goals are ensured. In the case of open systems, the engineer of one system does not control the other systems, which interact with it. Then, a relevant question from the point of view of this engineer is whether the agents from the other interacting systems, by ensuring their goals, necessarily have favorable side effect on its model. This is what we call *contribution*. Let us consider a last version of our example, CP_3.

Example 8 (CP_3). In this version, A has priority over B: when the provider provides resources in new_A at time t, A can get some of them, the remainder is sent to new_B and then at time $t + 1$, B can take some. In order to encode this, we introduce a new variable *even* marking the evenness of the current time unit (it is equal to 0 at even times and equal to 1 at odd times). A can act during transitions from even to odd time units (let us call them *even transitions*), and *provider* and B can act during odd transitions. Again, the provider is able to produce up to 20 units of the resource per odd transition, its goal $g_{provider}$ is changed into providing at least 18 units of the resource per time unit and g_A and g_B are both assigned to $\{A, B\}$ in the assignment $\mathcal{A}_3$. (Fig. 1 gives an overview of the goals and the assignments in the three versions of our example.)

In this version, whatever *provider* does, if by doing so it ensures the satisfaction of $g_{provider}$ then it provides at least 18 units of the resource per time unit, enabling A and B to ensure the satisfaction of g_A and g_B. We say that $g_{provider}$ *globally contributes* to g_A and g_B and we write $Contr_{\mathcal{A}_3}(g_{provider}, \{g_A, g_B\})$.

Again, one can also define, similarly, a relation of *local contribution*, that we do not detail here.

CP_1	$[\![g_A]\!] \triangleq \mathbf{XX}(\Box(obt_A \geqslant 6))$ $\mathcal{A}_1(g_A) = \{provider, A\}$	$[\![g_B]\!] \triangleq \mathbf{XX}(\Box(obt_A \geqslant 12))$ $\mathcal{A}_1(g_B) = \{provider, b\}$	
CP_2	$[\![g_{provider}]\!] \triangleq \mathbf{X}(\Box(new_A + new_B \geqslant 16))$ $\mathcal{A}_2(g_{provider}) = \{provider\}$	$[\![g_A]\!] \triangleq$ cf. CP_1 $\mathcal{A}_2(g_A) = \{A\}$	$[\![g_B]\!] \triangleq$ cf. CP_1 $\mathcal{A}_2(g_B) = \{B\}$
CP_3	$[\![g_{provider}]\!] \triangleq \mathbf{X}(\Box(new_A \geqslant 18))$ $\mathcal{A}_3(g_{provider}) = \{provider\}$	$[\![g_A]\!] \triangleq$ cf. CP_1 $\mathcal{A}_3(g_A) = \{A, B\}$	$[\![g_B]\!] \triangleq$ cf. CP_1 $\mathcal{A}_3(g_B) = \{A, B\}$

Fig. 1: Goals and assignments in the *client-provider* example

4 Formal Analysis

In this section we introduce the formal framework that we use to check the correctness criteria. Basically, given a specification K conforming to the Kore framework, checking a criterion consists in knowing whether goals in K are assigned to coalitions of agents able to ensure them.

Our approach consists of reducing such a question to a model-checking problem: *does a model $\mathcal{G}$ (in the logical sense) satisfy a formula φ ?* where: (1) the model of the possible behaviors of the system is derived from the description of agents and context properties; and (2) the formula expresses that some coalition(s) is (are) able to ensure some goal(s).

To achieve this, a logic that allows to reason about the ability of agents to ensure temporal properties is required. This is the aim of *temporal multi-agent logics*, such as ATL [AHK02], Chatterjee, Henzinger, and Piterman's Strategy Logic (SL) [CHP10] or

USL (Updatable Strategy Logic) [CBC13, CBC15], which is strictly more expressive than the former two, and which we originally proposed to address such issues.

One of the main specific features of USL is that it enables us to express situations where agents may be part of several different interacting coalitions. So, agents in USL can compose their behavior according to the different goals assigned to these coalitions. For this reason, we rely on USL in the following.

In Sect. 4.1, we briefly present USL. Then in Sect. 4.2, we present the reduction of our correctness criteria to instances of the model-checking problem for USL.

4.1 A temporal multi-agent logic for the formalization of Kore: USL

Let us first introduce the semantic concepts that are used in USL. Due to space constraints, we refer the reader to [CBC15] for a complete exposition of USL.

Semantic concepts Formulas of USL are interpreted in *concurrent game structures* (CGS), introduced in [AHK02] and then subsequently used with slight modifications in numerous works [BDCLLM09, DLLM10, MMV10, MMPV14][4]. Intuitively, a CGS is an extension of labelled transition systems dedicated to modelling multi-agent systems. In these systems, transitions are determined by the *actions* that *agents* perform. More precisely, at each state of any execution, each agent plays an action so that a transition is determined.

Definition 6. *A CGS is a tuple $\mathcal{G} = \langle \mathrm{Ag}, St, s_0, At, v, Act, tr \rangle$ where :*

- Ag *is a non-empty finite set of agents,*
- *St is an non-empty enumerable set of states,*
- *$s_0 \in St$ is the initial state,*
- *At is a non-empty finite set of atomic propositions*
- *v: $St \to \mathcal{P}(At)$ is a valuation function, to each state s it associates the set of atomic propositions that are true in s,*
- *Act is an non-empty enumerable set of actions,*
- *Let $Dec = Act^{\mathrm{Ag}}$ be the set of decisions,* i.e. *the set of total functions from the agents to the actions. Then $tr : St \times Dec \to St$ is the transition function: it decides the successor of a state, given this state and the set of actions played by the agents.*

The semantics of USL in CGSs is given by plays in a game, that are infinite sequences $(s_0, \delta_0) \cdot (s_1, \delta_1) \ldots$ where for each $k \in \mathbb{N}$:

- s_k is a state and δ_k is a decision
- $tr(s_k, \delta_k) = s_{k+1}$

In such a game, every agent plays w.r.t. a *multi-strategy*. A multi-strategy is a function from St^* to $\mathcal{P}(Act)$: given the current history of the game, it gives a set of possible actions for any agent following it. The datum of one multi-strategy per agent in the game is called a *multi-strategy profile*.

[4] As USL builds upon SL, our definition for CGS is the one from [MMV10, MMPV14].

During the evaluation of an USL formula in a CGS, the data concerning the different multi-strategies played by the agents are stored in a *context* κ. In a context, an agent may be bound to *several* multi-strategies, which is a particularity of USL in the field of multi-agent logics. As we will see below, USL also makes use of *multi-strategy variables*. Then, a context also contains information about the instantiations of multi-strategy variables to multi-strategies. We also use the notion of *multi-strategy profile* for a coalition of agents, which consists of one multi-strategy per agent in the coalition.

Given a context κ binding agents and variables to multi-strategies in a CGS, and given a state s of this CGS, we can define the notion of *outcomes* of s and κ. It is the set of executions that are possible in the CGS from s if each agent plays only actions that are allowed by all the multi-strategies she is bound to in κ.

Syntax and semantics of USL Let us now present the syntax of USL and an intuition of its semantics. We start by defining USL pseudo-formulas. We distinguish between state pseudo-formulas (interpreted on states, whose operators deal with multi-strategies quantification and binding of multi-strategies to agents) and path pseudo-formulas (which express temporal properties).

Definition 7 (Pseudo-formulas of USL). *Let* Ag *be a set of agents, At a set of propositions, and X a set of multi-strategy variables. Then, the set of* USL (Ag, At, X) pseudo-formulas *is generated by the following grammar (with $p \in At$, $x \in X$ and $A \subseteq$ Ag):*

- State pseudo-formulas: $\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\!\langle x \rangle\!\rangle\varphi \mid (A \rhd x)\psi \mid (A \not\rhd x)\psi$
- Path pseudo-formulas: $\psi ::= \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \mathsf{X}\,\psi \mid \psi\mathbf{U}\psi$

Then, well formed formulas are pseudo-formulas where every quantified multi-strategy variable is fresh w.r.t. the scope in which it is introduced. Formally:

Definition 8 ((Well formed) formulas). *A pseudo-formula φ is a* (well formed) formula *iff for any sub-formula $\langle\!\langle x \rangle\!\rangle\varphi'$ of φ and for any sub-formula $\langle\!\langle y \rangle\!\rangle\varphi''$ of φ', x and y are distinct variables.*

Here we do not detail the definition for the relation of semantic satisfaction for USL ($\models_{\mathrm{USL}}$). We just give an intuition for the main operators:

- The operator $\langle\!\langle x \rangle\!\rangle$ is an existential quantifier over multi-strategies: a formula $\langle\!\langle x \rangle\!\rangle\varphi$ is true in a state s of a CGS $\mathcal{G}$, under context κ, iff there is a multi-strategy σ s.t. the formula φ is true in s and $\mathcal{G}$ under the context κ enriched by the fact that x is instantiated by σ.
- The operator $(A \rhd x)$ is a binding of agents in A to the multi-strategy instantiating x in the current context: a formula $(A \rhd x)\psi$ is true in a state s of a CGS $\mathcal{G}$, under context κ, iff the formula ψ is true in any execution in the outcomes of s and $\kappa[A \oplus x]$, where $\kappa[A \oplus x]$ is the context κ enriched with the fact that agents in A are now bound to the multi-strategy instantiating x in κ (in addition to the multi-strategies they were already bound to in κ, if there are some). In USL, this set of outcomes may be empty (if there are agents bound to multi-strategies in empty intersection in the context). In such a case, the current execution stops.

- $(A \not\triangleright x)$ unbinds agents in A from the multi-strategy instantiating x in the current context: it is interpreted in a similar way as $(A \triangleright x)$, except that the binding of agents in A to x is deleted from the current context (instead of being added).
- The semantics of temporal operators follows the classical definition of their interpretation in possibly finite executions [EFH⁺03]. In the following, given a possibly finite execution λ and an integer i, we write $\lambda_{\geqslant i}$ for the sequence obtained from λ by deleting its i first elements:
  - A formula $\mathsf{X}\,\psi$ is true in λ iff λ contains at least two states and ψ is true in $\lambda_{\geqslant 1}$.
  - A formula $\psi_1 \mathbf{U} \psi_2$ is true in λ iff there is $i \in \mathbb{N}$ s.t. λ contains at least $i+1$ states, ψ_2 is true in $\lambda_{\geqslant i}$ and for all $0 \leqslant j < i$, ψ_1 is true in $\lambda_{\geqslant j}$.

4.2 Reduction of the assignment problem to the model-checking of USL

Using USL, we can reduce the satisfaction problem for the assignment correctness criteria from Sect. 3 to instances of model-checking. First, notice that from the description of agent capabilities in an instance K of Kore, and from the set of context properties for K, one can derive a CGS $\mathcal{G}_\mathsf{K}$. For the sake of simplicity, we do not detail this translation here. Basically:

- Agents of $\mathcal{G}_\mathsf{K}$ are those of K,
- the set of states in $\mathcal{G}_\mathsf{K}$ is the set of possible instantiations for the variables in K which respect the static context properties,
- the initial state of $\mathcal{G}_\mathsf{K}$ is chosen non-deterministically within those satisfying the initial context properties,
- the set of atomic propositions is the set of propositions in Cond that are used in K for the description of goals or for the description of agents capabilities,
- the valuation function is given by the natural evaluation of Cond formulas in variables instantiations,
- the transition function encodes the capacities of agents to change the value of variables, according to the description of their capabilities in K.

Then, thanks to USL formulas, we can express that coalitions of agents ensure the satisfaction of the different correctness criteria for K. The formalization of the criteria are given in Def. 9. It makes use of the following notations:

- $\overrightarrow{x}$ denotes a vector of multi-strategy variables (one can see it as a multi-strategy profile variable). Then, for any coalition of agents $A = \{a_1, \ldots, a_n\}$, the notation $(A, \overrightarrow{x})$ abbreviates the sequence $(a_1, x_{a_1}), \ldots (a_n, x_{a_n})$
- for a variable x, $\ulcorner x \urcorner$ is the universal quantifier over x. It is the dual operator of $\langle\!\langle x \rangle\!\rangle$. In other words, for any USL formula φ, $\ulcorner x \urcorner \varphi \triangleq \neg \langle\!\langle x \rangle\!\rangle \neg \varphi$.

Definition 9 (Formalisation of the correctness criteria in USL). *Let K be an instance of* Kore *with assignment $\mathcal{A}$. Let G be a set of goals in K and let g be a goal in K s.t.*

$g \notin G$. Then:

$$\begin{aligned}
[\![LC_{\mathcal{A}}(G)]\!] &\triangleq \bigwedge_{g \in G} (\langle\!\langle \overrightarrow{x_g} \rangle\!\rangle (\mathcal{A}(g) \rhd \overrightarrow{x_g})[\![g]\!]) \\
[\![GC_{\mathcal{A}}(G)]\!] &\triangleq \langle\!\langle \overrightarrow{x} \rangle\!\rangle (\bigwedge_{g \in G} (\mathcal{A}(g) \rhd \overrightarrow{x_g})[\![g]\!]) \\
[\![Coll_{\mathcal{A}}(g, G)]\!] &\triangleq \langle\!\langle \overrightarrow{x_g} \rangle\!\rangle (\mathcal{A}(g) \rhd \overrightarrow{x_g})([\![g]\!] \wedge [\![GC_{\mathcal{A}}(G)]\!]) \\
[\![Contr_{\mathcal{A}}(g, G)]\!] &\triangleq (\langle\!\langle y_g \rangle\!\rangle (\mathcal{A}(g) \rhd \overrightarrow{x_g})[\![g]\!]) \wedge \\
&\quad \Big(\ulcorner \overrightarrow{x_g} \urcorner \Big((\mathcal{A}(g) \rhd \overrightarrow{x_g})[\![g]\!] \to ((\mathcal{A}(g) \rhd \overrightarrow{x_g})[\![GC_{\mathcal{A}}(G)]\!]) \Big) \Big)
\end{aligned}$$

Thus, for any correctness criterion C from Sect. 3, and for any instance K of Kore, the satisfaction of C by K is formalized by the relation $\mathcal{G}_{\mathsf{K}} \models_{\text{USL}} [\![C]\!]$.

Let us discuss on the formulas in Def. 9.

- The assignment is locally correct iff for each goal g, there is a multi-strategy profile s.t., by playing it, the agents to which g is assigned can ensure its satisfaction. Hence, we check the satisfaction of this criterion by considering one (possibly different) multi-strategy profile per considered goal. Let us consider for instance CP_1 from Examples 3 and 5. We see that *provider* can play the multi-strategy $alwaysNew_A \geqslant 6$ (consisting in restricting to the choices of values for new_A and new_B s.t. $new_A \geqslant 6$ at any time of the execution) and, if *provider* does so, A can play $alwaysObt_A \geqslant 6$ so that g_A is ensured. Similarly, by playing respectively $alwaysNew_B \geqslant 12$ and $alwaysObt_A \geqslant 12$, *provider* and B can ensure g_B. So, $(\langle\!\langle \overrightarrow{x_{g_A}} \rangle\!\rangle (A, provider \rhd \overrightarrow{x_{g_A}})[\![g_A]\!]) \wedge (\langle\!\langle \overrightarrow{x_{g_B}} \rangle\!\rangle (B, provider \rhd \overrightarrow{x_{g_B}})[\![g_B]\!])$ is true: $\mathcal{G}_{CP_1} \models_{\text{USL}} [\![LC_{\mathcal{A}_1}(\{g_A, g_B\})]\!]$.
- For the global correctness, we consider one single multi-strategy profile, which imposes on each agent to act in a coherent way. The assignment is globally correct iff there is a multi-strategy profile $\overrightarrow{x}$ s.t. for each goal g, if the agents in the coalition $\mathcal{A}(g)$ play according to $\overrightarrow{x}$ (in the definition, we note $\overrightarrow{x_g}$ the part of $\overrightarrow{x}$ that concerns the agents in $\mathcal{A}g$), then they ensure the satisfaction of g. We can easily see that $[\![GC_{\mathcal{A}_1}(G)]\!]$ is not true in $\mathcal{G}_{CP_1}$. Indeed, *provider* cannot play a multi-strategy that satisfies both g_A and g_B at the same time. (According to its capabilities, *provider* cannot deliver more than 15 units of the resource at a time).
- To ensure that a goal g globally collaborates to a set of goals G, we need a multi-strategy profile $\overrightarrow{x}$ s.t., if followed by the agents in $\mathcal{A}(g)$, $\overrightarrow{x}$ ensures at the same time that:
  - g is satisfied
  - the evolution of the model is constrained in such a way that the assignment $\mathcal{A}$ becomes globally correct for the set of goals G.

  According to this definition, the example g_{prov} globally collaborates to g_A and g_B in CP_2 (see Sect. 3.3). Indeed, since it may produce up to 20 units of the resource per time unit, *provider* can play a multi-strategy that will allow both A and B to ensure their respective goal. Furthermore, recall that $\mathcal{A}_2(g_A)$ is reduced to $\{A\}$ and $\mathcal{A}_2(g_B)$ is reduced to $\{B\}$ and observe that CP_2 is not globally correct: to be able to ensure their goals, A and B depend on the multi-strategy played by *provider*.
- The contribution relation is an universally quantified variant of the collaboration : a goal g globally contributes to a set of goals G iff

- the agents in $\mathcal{A}(g)$ are able to ensure g,
- for any multi-strategy profile $\overrightarrow{x_g}$ that makes $\mathcal{A}(g)$ ensure g, $\overrightarrow{x_g}$ also makes $\mathcal{A}(g)$ constrain the evolution of the system in a way that G becomes globally correct.

In CP_2, by playing, for example, the multi-strategy consisting in setting new_A to 5 and new_B to 11, *provider* ensures its goal of producing at least 16 units per time unit, but it prevents A and B to ensure both goals g_A and g_B. On the other hand, in CP_3, the satisfaction of $g_{provider}$ (providing at least 18 units of the resource) by the provider allows A an B to ensure their goals, provided the new assignment of both g_A and g_B to $\{A, B\}$. So $[\![Contr_{\mathcal{A}_3}(g_{prov}, \{g_A, g_B\})]\!]$ is true in $\mathcal{G}_{CP_3}$.

Theorem 1. *The possible entailment relations between our correctness criteria for the assignment are given in the following figure, where arrows* $1, 2$ *and* 3 *represent a strict entailment relation, and the crossed out arrow* 4 *means that there is no entailment between* $GC_{\mathcal{A}}(G)$ *and* $Contr_{\mathcal{A}}(g, G\backslash\{g\})$*, in either direction. Each arrow should be read by universally quantifyfing the assignment* $\mathcal{A}$*, the set of goals* G *and any goal* $g \in G$*. For example:*

- ***Entailment*** *For any instance* $\mathcal{K}$ *of* Kore *with assignment* $\mathcal{A}$ *and for any subset* G *of goals in* $\mathcal{K}$*, for any* $g \in G$*, if* $\mathcal{G}_{\mathcal{K}} \models_{USL} [\![GC_{\mathcal{A}}(G)]\!]$ *then* $\mathcal{G}_{\mathcal{K}} \models_{USL} [\![Coll_{\mathcal{A}}(g, G\backslash\{g\})]\!]$.
- ***Strictness*** *It is not true that for any instance* $\mathcal{K}$ *of* Kore *with assignment* $\mathcal{A}$ *and for any subset* G *of goals in* $\mathcal{K}$*, for any* $g \in G$*, if* $\mathcal{G}_{\mathcal{K}} \models_{USL} [\![Coll_{\mathcal{A}}(g, G\backslash\{g\})]\!]$ *then* $\mathcal{G}_{\mathcal{K}} \models_{USL} [\![GC_{\mathcal{A}}(G)]\!]$.

$$\begin{tikzcd}
LC_{\mathcal{A}}(G) & & Coll_{\mathcal{A}}(g, G\backslash\{g\}) & \\
& GC_{\mathcal{A}}(G) \arrow[lu, "1"'] \arrow[ru, "2"] \arrow[rr, leftrightarrow, "4", "/" marking] & & Contr_{\mathcal{A}}(g, G\backslash\{g\}) \arrow[lu, "3"']
\end{tikzcd}$$

Proof (sketch). Each item in this proof sketch refers to the arrow in figure above that has the corresponding label.

1. Entailment : straightforward. Strictness: as seen in Sect. 4.2, CP_1 provides a counterexample.
2. Entailment: suppose there is a general multi-strategy profile $\overrightarrow{x}$ s.t., by playing it, every coalition ensures its goal. Then, by playing along $\overrightarrow{x}$, the agents in g ensure at the same time the satisfaction of g and the global correction of the model reduced to $G \setminus \{g\}$. Strictness: as seen in Sect. 4.2, CP_2 is a counterexample for the converse.
3. Entailment : straightforward. Strictness: goal $g_{provider}$ in CP_2 provides a counterexample: by playing multi-strategies $allwaysNew_A = 5$ and $allwaysNew_B = 11$, *provider* ensures $g_{provider}$ but prevents the satisfaction of g_A and g_B by the other agents.
4. Left to right: CP_3 satisfies $Contr_{\mathcal{A}_3}(g_{provider}, GC(\{g_A, g_B\}))$ but, to be able to ensure their goals, A and B depend on the satisfaction of $g_{provider}$ by *provider*, so $GC_{\mathcal{A}_3}(\{g_A, g_B\})$ is not true. Right to left: consider a minimal example where *provider* is able to provide 5 units of the resource per time unit, and is assigned both goals g_- to provide 2 units, and g_+ to provide 4 units. This model is globally correct but g_- does not contribute to $\{g_+\}$. □

5 Related Work

This work was initially developed in the context of Requirements Engineering [CBC11] and took inspiration from the state-of-the-art in this domain, in particular from Kaos [vL09, LVL02b, Let02]. In this method, goals are gradually refined until reaching so-called requirements. Then, agents are assigned the responsibility of realizing the latter by relying on operations (our agents are directly assigned goals to simplify the presentation). In some developments of Kaos, a notion of controllable and monitorable conditions [LVL02a, vL04] is used as a criterion of satisfiability of realization: an agent can perform an operation if it monitors the variables in its pre-conditions and controls the ones in its post-conditions. So, in Kaos and contrary to Kore, (1) capabilities are not conditioned by the state of the system; and (2) agents interactions are not analyzed.

Another important RE approach is Tropos [CDGM10a, CDGM10b, MS06, CS09]. In this line of work, a notion of *role* is introduced which gathers a set of specifications to be satisfied by the system. The two notions of agents and roles are then confronted. The adequacy between them is examined using propositional logic: roles are described through commitments and agents may ensure these depending on their capabilities. Considerations on time and interactions between agents are only led using natural language. Thus the method makes the verification of questions of the sort: "can agent a ensure transition tr?" possible. But the possible interactions between agents, as modifications of the common environment, are not considered formally. Kore precisely aims at unifying the multi-agent and behavioral aspects.

On the logical side, ATL and ATL* [AHK02] consider the absolute ability of coalitions to ensure propositions whatever other agents do. But there is no contextualization w.r.t. the strategies followed by different agents. More recently, this contextualization was considered for ATL* [BDCLLM09]. This proposition only uses implicit quantification over strategies for coalitions, preventing from considering different strategy quantifications for a given coalition. This problem was also tackled in SL [MMV10] where quantification over strategies is made using explicit variables. Our logic USL was first developed in [CBC13, CBC15]. Its syntax is inspired by that of SL, but it contains in addition treatment for the composition of several multi-strategies for a given agent.

6 Conclusion and Future Work

In this article, we proposed a framework to model the assignment of behavioral goals to agents, described with capabilities. Then, we addressed the evaluation of such an assignment, informally referred to as the *assignment problem*. We proceed in two steps:

- Rather than defining one criterion that would provide a unique "yes or no" answer, we think it is more relevant to define several correctness criteria, each involving a different level of interaction between agents. We compared these criteria through a logically defined entailment relation between them.
- We provide a formalization of the different criteria using a temporal multi-agent logic, USL, and we reduce the verification of these criteria to the model-checking problem of this logic.

As a future work, the relevance of new correctness criteria for the assignment could be investigated. A direction would be to develop a formal language dedicated to the specification of criteria, using the satisfaction of goals by the coalitions they are assigned to as atoms, and the relations between these goals as operators. Thanks to such a language, we could extend and refine the criteria that can be checked in Kore.

Another direction is to study dependence relations between the multi-strategies that are played by the different coalitions. In the contribution relation for example, a coalition A_1 is able to find a favorable multi-strategy profile, whatever another coalition A_2 does (because of the nesting of multi-strategy quantifiers). In other words, A_1 knows the whole multi-strategy profile chosen by A_2 when choosing its own multi-strategy profile, which is a very strong assumption. However, it is possible in USL to characterize several forms of independence of A_1's multi-strategy profile with respect to A_2's multi-strategy profile, so that this question could be integrated in the definition of new correctness criteria.

In [CBC15], we proved that the model-checking problem for USL is decidable, but does not support any elementary bound. Nevertheless, we have a strong conjecture stating that the restriction of USL to memoryless multi-strategies is decidable in PSPACE. Thus, restricting to memoryless multi-strategies appears as an important condition for a tractable use of our proposition. Then, research should be led in order to further characterize the class of systems for which a memoryless semantics is adequate.

Finally, it can happen that some goals are not fully achievable by the agents they are assigned to. Especially, so called *soft goals* don't have any clear cut satisfaction criterion. Then, considering and searching the best multi-strategies with regards to these goals would rise further analyses. Different notions of optima are indeed expressible in USL and could be used for defining different notions of optimal multi-strategies.

References

[AHK02] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[BDCLLM09] T. Brihaye, A. Da Costa Lopes, F. Laroussinie, and N. Markey. ATL with strategy contexts and bounded memory. *Logical Foundations of Computer Science*, pages 92–106, 2009.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, pages 203–236, 2004.

[CBC11] Christophe Chareton, Julien Brunel, and David Chemouil. A formal treatment of agents, goals and operations using alternating-time temporal logic. In *Brazilian Symposium on Formal Methods (SBMF)*, pages 188–203, 2011.

[CBC13] Christophe Chareton, Julien Brunel, and David Chemouil. Towards an Updatable Strategy Logic. In *Proc. 1st Intl WS on Strategic Reasoning SR*, 2013.

[CBC15] Christophe Chareton, Julien Brunel, and David Chemouil. A logic with revocable and refinable strategies. *Inf. Comput.*, 242:157–182, 2015.

[CDGM10a] A.K. Chopra, F. Dalpiaz, P. Giorgini, and J. Mylopoulos. Modeling and reasoning about service-oriented applications via goals and commitments. In *Advanced Information Systems Engineering*, pages 113–128. Springer, 2010.

[CDGM10b] A.K. Chopra, F. Dalpiaz, P. Giorgini, and J. Mylopoulos. Reasoning about agents and protocols via goals and commitments. In *Proc. of the 9th Intl Conf. on Autonomous Agents and Multiagent Systems-Volume 1*, pages 457–464, 2010.

[CHP10] Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. Strategy logic. *Inf. & Comp.*, 208(6):677–693, 2010.

[CS09] A.K. Chopra and M.P. Singh. Multiagent commitment alignment. In *Proc. of The 8th Intl Conf. on Autonomous Agents and Multiagent Systems-Volume 2*, pages 937–944, 2009.

[DLLM10] Arnaud Da Costa Lopes, François Laroussinie, and Nicolas Markey. ATL with strategy contexts: Expressiveness and model checking. In *IARCS Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 8, pages 120–132, 2010.

[EFH+03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Computer Aided Verification*, pages 27–39, 2003.

[Let02] E. Letier. *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD thesis, Université Catholique de Louvain, 2002.

[LVL02a] E. Letier and A. Van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proc. of the 24th Intl Conf. on Software Engineering*, pages 83–93. ACM, 2002.

[LVL02b] E. Letier and A. Van Lamsweerde. Deriving operational software specifications from system goals. In *Proc. of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, page 128. ACM, 2002.

[Mai98] Mark W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998.

[MMPV14] Fabio Mogavero, Aniello Murano, Giuseppe Perelli, and Moshe Y Vardi. Reasoning about strategies: On the model-checking problem. *ACM Transactions on Computational Logic (TOCL)*, 15(4):34, 2014.

[MMV10] Fabio Mogavero, Aniello Murano, and Moshe Y. Vardi. Reasoning about strategies. In *IARCS Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 8, pages 133–144, 2010.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

[MS06] A. Mallya and M. Singh. Incorporating commitment protocols into Tropos. *Agent-Oriented Software Engineering VI*, pages 69–80, 2006.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002.

[vL03] Axel van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures*, pages 25–43, 2003.

[vL04] Axel van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. In *ICSE*, pages 148–157, 2004.

[vL09] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[Yu96] Eric Siu-Kwong Yu. *Modelling strategic relationships for process reengineering*. PhD thesis, University of Toronto, Toronto, Ont., Canada, Canada, 1996. UMI Order No. GAXNN-02887 (Canadian dissertation).

[Yu09] Eric S. K. Yu. Social modeling and *i**. In *Conceptual Modeling: Foundations and Applications*, pages 99–121, 2009.

Hard-wiring CSP Hiding: Implementing Channel Abstraction to Generate Verified Concurrent Hardware

F. J. S. Macário[1] and M. V. M. Oliveira[1] *

Departamento de Informática e Matemática Aplicada, UFRN, Brazil

Abstract. Throughout the development of concurrent systems, complexity may easily grow exponentially yielding a very complex and error-prone process. By using formal languages like CSP we may simplify this task increasing the level of confidence on the resulting system. Unfortunately, such languages are not executable: the gap between the specification language and an executable program must be solved. In previous work, we presented a tool, csp2hc, that translates a considerable subset of CSP into Handel-C source code, which can itself be converted to produce files to program FPGAs. This subset restricts the use of data structures and CSP hiding. In this paper, we present an extension to csp2hc that includes sequences in the set of acceptable data structures and completely deals with the CSP hiding operator. Finally, we validate our extension by applying the translation approach to a industrial scale case study, the steam boiler.

Keywords: concurrency, CSP, Handel-C, abstraction, data structures, code synthesis.

1 Introduction

Concurrent systems encompass two or more components in progress at the same time [4]. They were initially programmed mainly to build operating systems. Nowadays, however, concurrency has been strongly used to develop more general applications. Throughout this development, however, it is very difficult to ensure the quality of the resulting system based solely on informal inspections. The system dimension together with the complexity added by concurrency have an important impact on this increasing difficulty. In the development of concurrent applications, ensuring the software correctness is even more complicated because developers have to deal with the parallel execution and synchronisation of many components. This complexity usually yields a complex and error-prone development process [19]. Moreover, there are many properties in the concurrent world that need to be taken into consideration like, for instance, deadlock, livelock, and multi-synchronisation. Communicating Sequential Processes, CSP [19],

* Partially supported by INES and CNPq (grants 573964/2008-4 and 483329/2012-6) and Instituto Metrópole Digital.

was proposed to deal with these properties more explicitly. Using CSP, we are able to specify systems and to refine them. Both, the specification and the refinement, can then be validated and verified using the Failures-Divergence Refinement model-checker FDR3 [22]. This approach ensures the system consistency and reliability [9].

CSP is a process algebra used to describe concurrent systems as components interactions [21]. It has been broadly used on specification and analysis of concurrent systems. CSP provides a good notation that allows specification to be clearly expressed and extended. Using this notation, the systems are abstracted to behaviours of unities, called processes, which are described by events, algebraic operators and other less complex processes. The use of CSP to specify concurrent systems is aided by tools like FDR3, whose input specifications are written in $\mathsf{CSP_M}$ [19], the machine-readable version of CSP. FDR3 is able to automatically verify the correctness of the specifications as well as other properties like determinism, deadlock freedom and divergence freedom.

The levels of abstraction (specification, design and implementation) can be easily described using CSP, but the translation of the implementation to a programming language is still required at the end of the refinement process, preferably to a language that supports the CSP concurrency style through channels, such as occam-2 [11] and Handel-C [13].

The process to translate formal specifications of a system to a programming language is usually a non-trivial process and is considered problematic. In [16], we presented a tool, csp2hc, that automatically translates from $\mathsf{CSP_M}$ to Handel-C. csp2hc accepts a subset of $\mathsf{CSP_M}$ operators that includes, `SKIP`, `STOP`, sequential composition, parallel composition (including interleaving), recursion, prefixing, external choice, internal choice, alternation, guarded processes, `IF-THEN-ELSE`, datatypes, constants, functions, and some expressions.

In this paper, we extend csp2hc, increasing the subset of accepted $\mathsf{CSP_M}$ by adding the CSP abstraction operator (hiding) and almost all sequence expressions: sequence literals (i.e. `<>` and `<1,4,5>`), closed range (`<m..n>`), sequence concatenation (`s^t`), the length of a sequence (`#s` and `length(s)`), testing if a sequence is empty (`null(s)`), the head of a non-empty sequence (`head(s)`), the tail of a non-empty sequence (`tail(s)`) and testing if a given object occurs in a given sequence (`elem(x,s)`). CSP hiding makes a given set of events internal, that is, the events become invisible to the environment and beyond its control. Hence, the hiding operator can be used to encapsulate the events within a process and to remove them from the interface. The translation of hiding is a non-trivial process, which is described in Section 3.

In its current version, csp2hc automatic translates many of the classical case studies in the literature. For instance, the Dinning Philosophers proposed by Dijkstra in 1965 [8]. In this paper, we present the translation process of a more complex case study, which formalises the control system of a Steam Boiler [7].

This paper is organised as follows. In Section 2 we introduce $\mathsf{CSP_M}$, Handel-C, and briefly describe the previous version of csp2hc. Section 3 describes one of the main contributions of this paper, the extensions added into csp2hc: the sequence expressions and the hiding operator. In Section 4 we present our case study and the steps performed to translate the CSP$_M$ specification of the Steam Boiler into Handel-C. Our conclusions and intended future work can be found in Section 5.

2 Background

In this section, we describe CSP, Handel-C, and csp2hc's previous version. Here, we focus on the features used in the context of this paper.

2.1 CSP

Communicating Sequential Processes (CSP) is a formal language for describing patterns of interaction. It is based on a mathematical theory, a set of proof tools, and an extensive literature [10,19,21,20]. Most of the CSP tools, like FDR3 and ProBE, accept a machine-processable CSP, called CSP$_M$.

The simplest process of all is `STOP`. This process is never prepared to engage in any of its interfaces events. Other simple process is `SKIP`, that indicates that the process has reached successful termination. A prefixing `c -> P`, pronounced "c then P", describes an object that first engages in the event `c` and then behaves as described by `P`. The process `P1[]P2` is an external choice between process `P1` and `P2`: it allows the environment to resolve the choice by communicating an initial event to one of the processes. When environment has no control over the choice (i.e. the choice is resolved internally), we have an internal choice, which is written `P1|~|P2`. The parallel composition `P1[|cs|]P2` synchronises `P1` and `P2` on the events in the synchronisation set `cs`; events that are not listed in `cs` occur independently. The interleaving `P1|||P2` runs the processes independently. The guarded process `g & P` behaves like `P` if the predicate `g` is true; it deadlocks otherwise. The renaming `P[[a<-b]]` behaves like `P` except that all occurrences of `a` in `P` are replaced by `b`. Finally, using the hiding operator `P\cs`, we may hide all events in `cs` from the environment.

In Figure 1, we illustrate CSP$_M$ by presenting the specification of a buffer. The specification contains special comments called directives (`--!!`), which give extra information to csp2hc, such as: direction of communication (input or output) of simple synchronisation channels; types of processes' arguments; the main behaviour of the system; the length of integers used in the system; the maximum size of sequences; channels that will behave like buses (communication with the environment); and the maximum size of sequences that the specification is using. The details about these directives will be described in Section 3.

The buffer receives the data via an input channel, `read`, which is defined as a bus. It is important to notice that buses are not part of CSP: they are used only by csp2hc to define which special channels make communications between the external environment and the system. After receiving the data, the buffer transmits it to the storage component via the internal channel `input`, which is hidden from the environment. In this specification, we use a sequence to store the

```
--!! mainp SYSTEM
--!! int_bits 3
--!! seq_max_size 6

channel input, output : Int
channel read, write : Int

--!! bus read
--!! bus write

--!! channel input in within B
--!! channel output out within B
--!! arg s integerSequence within B
--!! arg n integer within B
B(s,n) =
 #s > 0 & output!head(s) -> B(tail(s),n)
 []
 #s < n & input?x -> B(s^<x>,n)
```

```
--!! arg n integer within BUFFER
BUFFER(n) = B(<>,n)

BUSIN = read?x -> input!x -> BUSIN

BUSOUT = output?x -> write!x -> BUSOUT

BUSES = BUSIN ||| BUSOUT

SYSTEM =
    (BUFFER(4)
        [| {| input, output |} |]
    BUSES)\{|input,output|}
```

Fig. 1. CSP$_{\mathsf{M}}$ Example: a Buffer

data. The storage component has an external choice: it may either communicate the first element of the sequence and remove it from the sequence or receive a new element and add it to the sequence. Both options are guarded: an output may only be given if the sequence has elements (the size of the sequence is greater than zero) and an input may only be accepted if the buffer is not full (the size of the sequence is less than the size of the buffer). Finally, the storage component may communicate forward a value via an internal channel `output`. This value is communicated to the environment using the `write` channel.

The storage process `B` uses a sequence to store the data: operations like concatenation, head and tail are used to handle the data. The process `BUFFER` behaves as a buffer of size 4. The processes `BUSIN` and `BUSOUT` receive inputs and give outputs, respectively. Their interleaving corresponds to the behaviour of the process `BUSES`. The `SYSTEM` is the main process. It is defined as the parallel composition of the `BUSES` with the `BUFFER`. Despite being a simple example, this example was not accepted by the previous version of csp2hc. This is due to the use of both (i) sequences and (ii) hiding.

2.2 Handel-C

Handel-C is a high level programming language that can be used to program low level hardware and is generally used to program `FPGAs`. The syntax of Handel-C is similar to `C`, but adds new constructors for describing parallel behaviour like, for example, the definition of channels, parallel composition, sequence blocks, and choice among events.

Using Handel-C the developer is able to write sequential code, but he may also maximise performance of the resulting hardware by using the high level parallel constructors. The parallel construct `par{P; Q;}` executes instructions from `P` and `Q` in parallel. The parallel branches may communicate with each other via channels. The semantics of `par{P; Q;}` corresponds to the CSP alphabetised parallel `P [|`α`(P) ||` α`(Q)|] Q`, where α(`P`) and α(`Q`) denotes all communications of P and Q, respectively. The prialt statement selects one of the channels that are ready to communicate, and communicates via this channel. The only data type allowed in Handel-C is int, which can be declared with a fixed size.

The target of the Handel-C compiler is low level hardware. In practical terms, developers may achieve considerable improvements in performance when using parallelism, in which case, the parallel statements will be executed at the same time in two separate blocks of hardware. The execution flow of Handel-C parallel statements is divided at the beginning of the parallel block and each branch is executed concurrently. The execution flow regroups at the end of the block when each branch has completed its own execution. If any branch completes its execution before the other, it waits for the others before continuing.

By way of illustration, we present a simple BUFFER, illustrate in Figure 2, that receives an integer value through a channel input and outputs it through channel output.

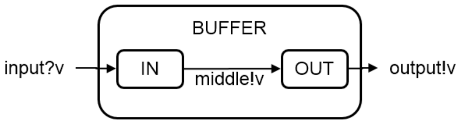


Fig. 2. A Simple Buffer

This buffer can be decomposed into a parallel branch IN that receives an integer value and passes it through channel middle to another parallel branch OUT that finally outputs this value. A possible CLIENT can interact with the BUFFER by sending an integer value via channel input and receiving it back via channel output. The Handel-C code presented below implements this interaction.

```
set clock = external "clock1";
chan int 8 input, output, middle;
void IN(){ int 8 v; while(1) { input?v; middle!v; } }
void OUT(){ int 8 v; while(1) { middle?v; output!v; } }
void BUFFER(){ par{ IN(); OUT(); } }
void CLIENT(){ int 8 v; input!10; output?x; }
void main(){ par { BUFFER(); CLIENT(); } }
```

We define an external clock named clock1, and declare the channels used in the system. The Handel-C function IN declares a local variable v and starts an infinite loop: in each iteration, it receives a value via channel input, assigns it to v, and writes its value on middle. The function OUT is very similar; however, it receives a value via middle and writes it on output. The BUFFER is defined as the parallel composition of IN and OUT. The main function is the parallel composition of the BUFFER with the CLIENT.

2.3 The Translator csp2hc

csp2hc fosters the use of the development methodology depicted in Figure 3. In this methodology, we start from an abstract CSP specification of a system (possibly centralised) and develop CSP an implementation (possibly distributed) using ProBE and FDR3 to prove its compliance with the specification. Finally, we use csp2hc to automatically translate the CSP implementation into Handel-C code, which can be validated using the DK simulator. Finally, the Handel-C code can be compiled and used to program FPGAs.

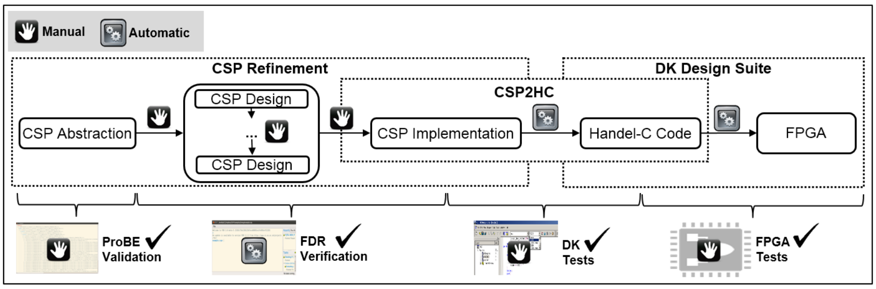


Fig. 3. Development Methodology

The automatic translation from $\mathsf{CSP_M}$ to Handel-C is simpler for some $\mathsf{CSP_M}$ constructs because Handel-C provides constructs that facilitate the description of concurrent behaviour based on CSP concepts. The version of csp2hc presented in [15] mechanised the translation of a subset of $\mathsf{CSP_M}$ to Handel-C, which included `SKIP`, `STOP`, sequential and parallel composition, recursion, prefixing, external and internal choice, alternation, guarded processes, renaming and interleaving, datatypes, constants, functions, and some expressions. But, the hiding operator and sequence expressions were not supported.

The implementation of sequence expressions is based on Handel-C `struct` construct. For each sequence operations available in CSP, we have implemented two Handel-C versions, one for signed integers and other for unsigned integers. Furthermore, the extension of csp2hc to accommodate hiding is not trivial because it requires the pre-processing of the $\mathsf{CSP_M}$ and the generation of a new $\mathsf{CSP_M}$ that refines the original input. In what follows, we described the details of both translation strategies.

3 Extensions to CSP2HC

The current version of csp2hc accepts a large subset of $\mathsf{CSP_M}$ constructs, which allows us to automatic translate many of the classical concurrent specifications available in the literature. In this paper, we present our contributions that increased that subset with sequence expressions and the $\mathsf{CSP_M}$ hiding. First, in Section 3.1 we describe how sequences were implemented in Handel-C. Next, we present the details of the implementation of hiding in Section 3.2.

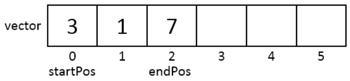


Fig. 4. Representation of sequence <3,1,7>

3.1 Data Structures: Sequence

CSP$_M$'s sequences have a standard behaviour. They can be used to store data in order and can be manipulated using CSP$_M$'s sequence operations: the closed range <n..m> returns the sequence with elements ranging from n to m; tail(s) returns the sequence resulting from removing the first element of s; complementary, head(s) returns the first element of s; elem(x,s) tests if a given x belongs to s; length(s) and #s return the size of s; null(s) tests if the sequence is empty; and, finally, s^t concatenates the two given sequences s and t. Three further CSP$_M$ sequence operations are available, but are not supported by our translation: the open range ..m, the distributed concatenation concat(s) and sequence comprehension. They are, however, on our research agenda.

The translation of sequences from CSP to Handel-C makes use of Handel-C structures defined below, which consist of three components: the start position startPos, the final position endPos, and the vector array that stores the data. The array has a fixed sized, which is informed by the user using a directive in the input CSP$_M$ specification and the components startPos and endPos are used as low water and high water marks that defines the elements of the array that are actually part of the sequence. The Handel-C code generated by csp2hc has two sorts if integers, signed and unsigned. The former is used to implement CSP$_M$ integers and the latter is used to represent values of existing datatypes. For this reason, since array types must be defined in advance, our translation defines two sorts of sequences. The structure integerSeqType is used for sequences of signed integers and the structure nonIntegerSeqType is used for sequences of unsigned integers.

```
struct integerSeqType{
    unsigned int 3 startPos; unsigned int 3 endPos; integer vector[6];
};
typedef struct integerSeqType integerSequence;

struct nonIntegerSeqType{
    unsigned int 3 startPos; unsigned int 3 endPos; unsigned int 3 vector[6];
};
typedef struct nonIntegerSeqType nonIntegerSequence;
```

In Figure 4, we have the representation of the sequence <3,1,7>.

For each CSP sequence operation, our translation has either a corresponding macro (for length, null and head) or a corresponding inline function (for tail and concatenation) that implements the original CSP$_M$ operation. For example, the CSP$_M$ functions head and tail are implemented as follows.

```
macro expr integerhead(S) = (S.vector[S.startPos]);
```

```
inline integerSequence integertail(integerSequence S){
    unsigned int 3 i; integerSequence S1;
    S1.startPos = 0; S1.endPos = S.endPos - 1; i = S.startPos + 1;
    while (i < S.endPos){
        S1.vector[i-1] = S.vector[i]; i = i + 1;
    }
    return S1;
}
```

The head of a sequence is simply the element at position `startPos` of the array. The function `integertail` builds a sequence that starts at position `startPos + 1` of the given sequence `S` and returns this sequence.

The translation of sequences required the creation of a new directive that provides further information to csp2hc. The directive `--!! seq_max_size n` informs csp2hc the maximum length `n` of sequences within the specification and is used to determine the length of the array in the Handel-C structure.

3.2 Hiding

CSP hiding encapsulates one or more events, turning these events internal to the process and inaccessible to the environment. Hence, the environment is unaware of the existence of hidden events because these events are removed from the interface. As an example, `P \ {|c|}` makes all events on `c` performed by `P` hidden from the environment. In our previous version of csp2hc, the "translation" of hiding consisted in ignoring it under certain conditions. These conditions guaranteed that only channels on which a communication between two parallel branches should take place were hidden. If these conditions were not satisfied, the translation would simply fail and the user would receive an error message. The conditions were:

H1 Hiding is only applied to communicating parallel processes: for every parallel composition `P [| cs |] Q` we increment a counter. During the analysis, channels may not be hidden if this counter is zero, meaning that we have no parallel branches. For example, this restriction forbids the translation of `(c -> SKIP) \ {| c |}`, which is deadlock free but whose translation would deadlock if we simply ignore the hiding.

H2 Hiding is only applied to channels on which communication must take place: during the analysis of the processes, hidden channels of a parallel branch must be in the synchronisation channel set of the parallel composition. Furthermore, they are either written on this branch and read on the parallel branch or read on this branch and written on the parallel branch. For reasons that are similar to those of the previous item, this restriction forbids the translation of `((c -> SKIP) ||| (b -> SKIP)) \ {| c |}`, which is also deadlock free but whose translation would deadlock if we simply ignore the hiding.

H3 Ignoring the hiding does not enable unallowed communications: we avoid communications that should not happen in the specification, by checking that no hidden channels on a parallel branch are in the synchronisation

channel set and in the alphabet of the other parallel branch. For example, this restriction forbids the translation of
`((c!1 -> SKIP) \ {| c |}) [| {|c|} |] (c?x -> SKIP)`,
on which there is no communication but whose translation would present a communication if we simply ignore the hiding.

If any of the above conditions were not satisfied, the translation would not succeed. In this paper, we remove this restriction by dealing with the cases on which the conditions were not satisfied. Our approach consists in refining the CSP_M specifications that do not satisfy the conditions above in an automatic way and translating these refinements into Handel-C. We have used FDR3 to verify the validity of the refinements in the failures-divergences model, ensuring that their behaviours are in accordance with the original specifications [19]. Since the refinements are a little more complex, the strategy is only applied in cases on which at least one of the above conditions fails. If, however, they are all satisfied, we simply ignore the hiding providing a simpler translation.

Basically, the strategy is to rename hidden channels and to create, for each application of the hiding to a branch, an auxiliary process `RUN_i` that continuously offer the renamed channels. The auxiliary process is executed in parallel with the renamed version `BODY_i` of the original branch. This parallel composition avoids the deadlocks that would otherwise take place because the translation of the original branches write on the renamed channels but no other branch reads from them. By way of illustration, let us consider the CSP_M process below.

```
P = ((a -> b -> SKIP) \ {|a|}) [| {|a, b|} |] ((b -> a -> SKIP) \ {|b|})
```

First, we create new channels for each of the hidden channels. Furthermore, for each of these channels, we also create an auxiliary channel `tau_terminate_n` whose function is described below. In our example, we have four new channels: `tau_a_1` and `tau_terminate_1`, which are related to the hiding of `a` in the left branch and `tau_b_1` and `tau_terminate_2`, which are related to the hiding of `b` in the right branch. The channels are grouped in the channel sets that are related to each use of hiding.

```
channel tau_a_1, tau_b_1, tau_terminate_1, tau_terminate_2

CT_1 = {| tau_a_1, tau_terminate_1 |}
CT_2 = {| tau_b_1, tau_terminate_2 |}
```

Next, for each branch to which a hiding is applied, we create an auxiliary process that recursively offers the newly created channels until it is terminated using the newly created `tau_terminate_n` channel.

```
RUN_1 = tau_a_1 -> RUN_1 [] tau_terminate_1 -> SKIP
RUN_2 = tau_b_1 -> RUN_2 [] tau_terminate_2 -> SKIP
```

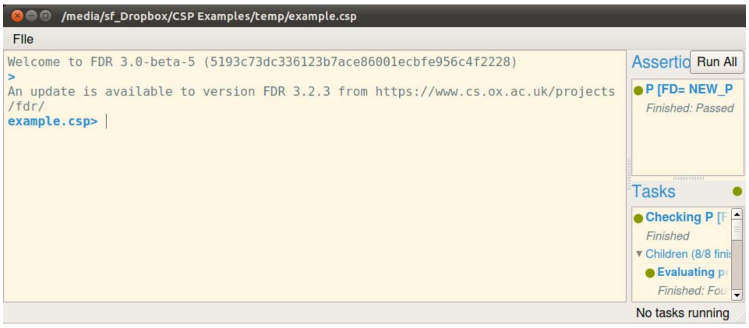
The body of the branch is changed to a process that replaces the hidden channels with the newly created ones and communicates on the `tau_terminate_n` channel that is related to that branch. In our example, we have the processes `BODY_1` and `BODY_2`, which correspond to the left and right branches of the parallel composition above, respectively.

```
BODY_1 = tau_a_1 -> b -> SKIP;tau_terminate_1 -> SKIP
BODY_2 = tau_b_1 -> a -> SKIP;tau_terminate_2 -> SKIP
```

The refinement process NEW_P is defined as the parallel composition of the rewritten version of the branches. Each branch is rewritten as the parallel composition of its new body with the corresponding RUN process.

```
NEW_P = (BODY_1 [|CT_1|] RUN_1) \CT_1 [|{|a,b|}|] (BODY_2 [|CT_2|] RUN_2) \CT_2
```

Using FDR3, we can verify that the new process NEW_P is indeed a refinement of the original process P (see Figure 5).



```
Welcome to FDR 3.0-beta-5 (5193c73dc336123b7ace86001ecbfe956c4f2228)
>
An update is available to version FDR 3.2.3 from https://www.cs.ox.ac.uk/projects
/fdr/
example.csp>
```

```
● P [FD= NEW_P
  Finished: Passed
```

```
● Checking P [F
  Finished
▼ Children (8/8 fini
  ● Evaluating p
    Finished: Fou
```

Fig. 5. FDR3 Verification of Hiding

During the translation process, csp2hc internally creates a new specification using NEW_P, the new channels and sets defined above, replaces the previous processes by their corresponding newly created ones, reprocesses this new specification, and finally translates it into Handel-C. This whole process is transparent to the user. The refined specifications satisfy the original restrictions on the hiding operator (conditions **H1-H3**). Hence, csp2hc simply ignores the resulting hiding, but preserves its original behaviour.

This process of translation and refinement verification in FDR3 was done for a test bench containing different specifications that exercises all possible combinations of using parallel composition, communication, and hiding. Furthermore, we also exercised this strategy with different case studies containing hiding on which at least one of the conditions **H1-H3** was not satisfied. The next section presents the development of one of these case studies, the Steam Boiler.

4 Case Study

The specification of a steam boiler control was first proposed by Bauer in 1993 [3]. It has been used as a case study for thorough comparison between the various design formalisms proposed in the literature. As a result, specifications of the steam boiler have been proposed in a large number of different formal languages.

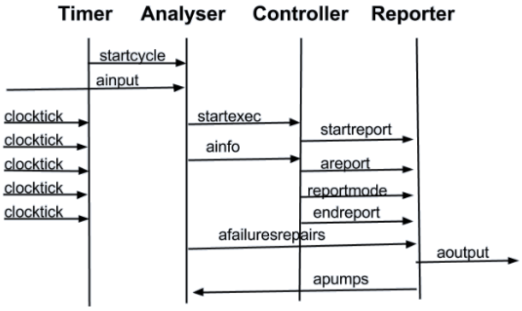


Fig. 6. Message sequence of Steam-Boiler

The problem consists in programming the control system of a steam boiler that can be found in power stations. The software that controls the steam boiler has to coexist with a physical environment with different elements: the steam boiler itself, a sensor to detect the level of water in the boiler, four pumps that supply water to the steam boiler, four pump controllers, a sensor that measures the amount of steam being produced, an operator's desk and a message transmission system.

At the moment the pump is turned on, a balancing pressure process is executed before the water is pumped into the boiler. This process takes five minutes. It may, however, be stopped instantaneously at any time. The controller that is responsible for the pump reports whether there is water passing through the pump. The program communicates with the external environment through messages that are transmitted by the reporter (See Figure 6).

The steam boiler program control operates in five different modes. The *initialisation* mode ensures that the water level is within the normal operating limits, and checks that the water and steam sensors are operating correctly. When operating in the *normal* mode, the program tries to keep the water level within the operating limits. In case the water sensor has not failed, but any other non-vital piece of equipment has failed, the program operates in *degraded* mode. In such cases, however, the system continues its execution. If the water sensor has failed and the program continues to operate, it operates in *rescue* mode. Finally, the program control enters in the *emergency stop* mode if any of the following happen:

- The program stops;
- The water level is near to either of the operating limits;
- A vital piece of equipment has failed;
- There is some irregularity in the protocol between the program and the physical equipment.

In [7], Freitas proposes an specification of steam boiler using *Circus*, a formal language that combines CSP, Z, and commands from Dijkstra's command language. Here, we adapted the *Circus* solution by removing the Z components and transforming the state components into process arguments. In what follows, we focus on the structure of the resulting CSP specification. The complete CSP specification, which includes all processes and csp2hc's directives can be found at the project's website[1].

The solution consists of four processes operating in parallel. The process formalises the `Timer` and ensures that the program's cycle begins at every five seconds.

```
TCycle(time) = (if (time + 1) % cycletime == 0
               then startcycle -> SKIP else SKIP);
               clocktick -> TCycle((time + 1) % cycletime)
Timer = TCycle(5)
```

The second process is the `Analyser`. It receives messages from the external environment and analyses the contents of these messages. After the analysis has completed, it sends an information service to the `Controller` process.

```
Analyser = startcycle -> ainput -> startexec -> InfoService
InfoService = ainfo -> InfoService
               [] afailuresrepairs -> apumps -> Analyser
TAnalyser = (Timer [| TAnalyserInterface |] Analyser) \TAnalyserInterface
```

The `Controller` decides the next actions to be taken based on the information that it receives. It is responsible for sending information to the `Reporter` process.

```
Controller = startexec -> startreport -> NewModeAnalysis;
             getmode?m -> (if m != emergencyStop
                           then |~| i: {0 .. limit} @ PutReports(i)
                           else SKIP);
             endreport -> Controller
...
TAController = (TAnalyser
                 [| TAControllerInterface |]
                 (Controller [|ModeStateInterface|] ModeState) \ ModeStateInterface
                ) \ TAControllerInterface
```

Finally, the `Reporter` is responsible for indicating the end of the cycle after reporting service to the `Controller`, packing them together with the outputs of the `Controller` and dispatching the results to the external environment.

```
Reporter = startreport -> ReportService
ReportService = [] m: NonEmergencyModes @ reportmode.m -> putmode!m -> ReportService
                [] areport -> ReportService
                [] reportmode.emergencyStop -> putmode!emergencyStop -> TidyUp
                [] TidyUp
TidyUp = endreport -> afailuresrepairs -> getmode?m -> aoutput!m -> apumps -> Reporter
TACReporter = (TAController
                [| TACReporterInterface |]
                    (Reporter [| ModeStateInterface |] ModeState) \ ModeStateInterface
               ) \ TACReporterInterface
SteamBoiler = TACReporter
```

Although based on the *Circus* specification proposed in [7], the resulting CSP specification was further refined to make it acceptable by csp2hc. For example, the replicated external choice used in the `Reporter` was refined by expanding them to an external choice resulting from instantiating the indexing variables. Furthermore, csp2hc does not accept mutually recursive processes in parallel compositions. For this reason, `Reporter` was also refined to remove the mutual recursion with `ReportService` and `TidyUp` as follows.

[1] http://www.dimap.ufrn.br/~marcel/research/csp2hc/

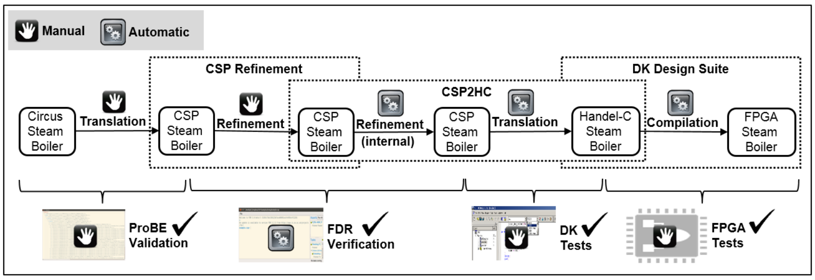


Fig. 7. Development Methodology

```
ReporterNew(n) =
    if (n == 0) then startreport -> ReporterNew(1)
    else if (n == 1) then
            reportmode?initialisation -> putmode!initialisation -> ReporterNew(1)
            [] reportmode?normal -> putmode!normal -> ReporterNew(1)
            [] reportmode?degraded -> putmode!degraded -> ReporterNew(1)
            [] reportmode?rescue -> putmode!rescue -> ReporterNew(1)
            [] areport -> ReporterNew(1)
            [] reportmode?emergencyStop -> putmode!emergencyStop -> endreport ->
               afailuresrepairs -> getmode?m -> aoutput!m -> apumps -> ReporterNew(0)
            [] endreport -> afailuresrepairs -> getmode?m -> aoutput!m ->
               apumps -> ReporterNew(0)
        else SKIP
ReporterR = ReporterNew(0)
```

The case study development methodology depicted in Figure 7 is based on the general development methodology depicted in Figure 3, but extends it by: (1) including an early translation from *Circus* to CSP (validated using FDR3's process explorer, ProBE); (2) instantiating the manual CSP refinement by a single refinement (verified using FDR3); and, finally, (3) including an automatic (and internal to csp2hc) CSP refinement to make the transformations described in this paper (transformation strategy validated using FDR3).

Overall, the final specification contained 21 processes specified on a 300 lines file. This refined CSP$_{\text{M}}$ specification of the steam boiler was finally given to csp2hc, which translate it into a 1466 lines Handel-C code in 3755ms[2].

5 Conclusions

The development of concurrent systems is inherently complex. The use of formal methods like CSP yields benefits to the development process by providing simpler means to specify and verify such systems. The work presented in this paper fosters the use of a development methodology in which developers: (1) specify the system's desired concurrent behaviour; (2) gradually refine the specification into a CSP implementation and verify the correctness of each refinement and

[2] Experiment Environment: Dell Inspiron 3000; Windows 8.1 x64; Intel Core I5 2.2 GHz with 3MB Cache; 8GB DDR3 RAM.

other properties using tools like FDR3, and; (3) automatically translate the $\mathsf{CSP_M}$ implementation into Handel-C, which can itself be converted to produce files to program FPGAs; (4) compile the resulting Handel-C code into VHDL using the DK Design Suite[3], and finally; (5) load this VHDL into a FPGA.

In this paper, we extend the tool that automatically translates from $\mathsf{CSP_M}$ to Handel-C. The previous version of csp2hc accepted a reasonable subset of CSP's operators that allowed users to translate most of the problems presented in the literature. Our extension, however, allow users to make use of sequences (and most of their operators) and CSP hiding. This contribution empowers the translation capabilities of csp2hc allowing it to translate more complex problems like our case study, the steam boiler.

5.1 Related Work

Handel-C's approach differs from BlueSpec's one [2]. The later is based on Verilog, a hardware description language that is useful for developing complex, bespoke hardware, exploiting a hardware engineer's skill and knowledge of circuits. The former is a programming language for compiling programs into hardware images of FPGAs or ASICs; it provides fast development and rapid prototyping, without hardware skills, and allows massive parallelism to be easily exploited.

The translation of process algebras into programming languages has already been considered in the literature. For instance, the refinement of $\mathsf{CSP_M}$ specifications into occam-2 and Ada 9X [1] code was presented in [11], which simply illustrates the translation without providing tool support. On the other hand, in [18], an automatic translation of $\mathsf{CSP_M}$ into C and Java is presented for a small subset of $\mathsf{CSP_M}$. They make use of libraries that provide models for processes and channels and allows programmers to abstract from basic constructs of these languages (i.e., JCSP [23] for Java and CCSP [12] for C). Furthermore, in previous work [14,6], we provided an automatic translation from a subset of *Circus* [5], a combination of CSP with Z [24] and Dijkstra's command language, into JCSP. This tran

Most of the translations between $\mathsf{CSP_M}$ and a programming language available in the literature target the generation of software. In [11], occam-2, which is the native programming language for a line of transputer microprocessors, is the target language. Unfortunately, it is not supported by any tool. In the literature, as far as we know, only [17] presents a tool that automatically converts a subset of $\mathsf{CSP_M}$ into Handel-C code. Their methodology is very similar to ours, but the subset of $\mathsf{CSP_M}$ considered is relatively small. Besides the subset considered in [17], csp2hc is able to translate complex communications, interleaved events, multi-synchronisation, and hiding. Furthermore, besides integers, we allow the use of sets and sequences.

[3] http://www.mentor.com/products/fpga/handel-c/dk-design-suite/

5.2 Future Work

Our translation of sequences is effectively a form of data refinement. Our research agenda includes proving that the transformation proposed in this paper is indeed a valid data refinement.

Further extensions to csp2hc will make it into a more powerful tool support for our methodology. First, the ability to translate the whole set of $\mathsf{CSP_M}$ operators and the data structures and operations of the functional language provided by FDR3 may encourage a large scale use of our methodology. Furthermore, some directives used to help in the process of translation may be inferred. For instance, the types of the processes arguments can be inferred based on a static analysis; hence, the need for its corresponding directive may be removed.

Finally, a thorough analysis of the performance of the resulting VHDL running in a real FPGA is our next step. This may instigate a deeper analysis of the translation decisions made during our research. This investigation may also foster a new (and perhaps more interesting) direction of research in which the $\mathsf{CSP_M}$ is directly translated into VHDL that may (or may not) have a more optimised behaviour.

References

1. A. Burns and A. Wellings. *Concurrency in Ada.* Cambridge University Press, Second edition, Nov 1997.
2. Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification invited talk. In *MEMOCODE '03: Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03)*, page 249, Washington, DC, USA, 2003. IEEE Computer Society.
3. J. C. Bauer. "Specification for a software program for a boiler water content monitor and control system". Institute of Risk Research, University of Waterloo, 1993.
4. Clay Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications.* O'Reilly Media, Inc., 2009.
5. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, **15**(2–3):146–181, 2003.
6. A. Freitas and A. L. C. Cavalcanti. Automatic Translation from *Circus* to Java. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *14th International Symposium on Formal Methods*, volume **4085** of *Lecture Notes in Computer Science*, pages 115–130. Springer, Aug 2006.
7. L. Freitas. Circus Example – Parsable Steam Boiler. Academia.edu, 2006.
8. Armando R. Gingras. Dining philosophers revisited. *SIGCSE Bull.*, 22(3):21–ff., August 1990.
9. Anthony Hall. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19, September 1990.
10. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
11. M. G. Hinchey and S. A. Jarvis. *Concurrent Systems: Formal Development in CSP.* McGraw-Hill, Inc., New York, NY, USA, 1995.
12. B. McMillin and E. Arrowsmith. CCSP-A Formal System for Distributed Program Debugging. In *Proceedings of the Software for Multiprocessors and Supercomputers, Theory, Practice, Experience*, Moscow - Russia, Sep 1994.

13. Mentor Graphics. *Handel-C Synthesis Methodology*, 2012.
14. M. V. M. Oliveira and A. L. C. Cavalcanti. From *Circus* to JCSP. In J. Davies *et al.*, editor, *Sixth International Conference on Formal Engineering Methods*, volume **3308** of *Lecture Notes in Computer Science*, pages 320–340. Springer-Verlag, 2004.
15. M. V. M. Oliveira, I. S. Medeiros Júnior, and J. C. P. Woodcock. A verified protocol to implement multi-way synchronisation and interleaving in csp. In RobertM. Hierons, MercedesG. Merayo, and Mario Bravetti, editors, *Software Engineering and Formal Methods*, volume 8137 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2013.
16. M. V. M. Oliveira and J. C. P. Woodcock. Automatic Generation of Verified Concurrent Hardware. In M. M. Larrondo-Petrie M. Butler, M. Hinchey, editor, *9th International Conference on Formal Engineering Methods*, volume 4789 of *Lecture Notes in Computer Science*, pages 286 – 306. Springer-Verlag, November 2007.
17. J. D. Phillips and G. S. Stiles. An Automatic Translation of CSP to Handel-C. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 19–38, sep 2004.
18. V. Raju, L. Rong, and G. S. Stiles. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 63–81, sep 2003.
19. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
20. A.W. Roscoe. *Understanding Concurrent Systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.
21. Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.
22. Alexandre Boulgakov-A.W. Roscoe Thomas Gibson-Robinson, Philip Armstrong. FDR3 — A Modern Model Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.
23. P. H. Welch. Process oriented design for Java: concurrency for all. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 51–57. CSREA Press, June 2000.
24. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.

Refinement strategies for Safety-Critical Java

Alvaro Miyazawa[1] and Ana Cavalcanti[2]

[1] alvaro.miyazawa@york.ac.uk,
[2] ana.cavalcanti@york.ac.uk,
Department of Computer Science
University of York, York, UK

Abstract. Safety-Critical Java (SCJ) is a version of Java that supports the development of real-time, embedded, safety-critical software. SCJ introduces abstractions that enforce a simpler architecture, and simpler concurrency and memory models, to support easier certification. In this paper, we detail a refinement strategy that takes a state-rich process algebraic design specification that adheres to a cyclic executive pattern and produces an SCJ design that can be automatically translated to code. We then show how this refinement strategy can be extended to support more complex patterns that include non-terminating and multiple missions.

1 Introduction

Safety-Critical Java (SCJ) [7] is a version of Java suitable for the development of verifiable real-time software. It incorporates part of the Real-Time Specification for Java (RTSJ) [17], introduces new abstractions such as Safelets and Missions, and removes garbage collection by enforcing the use of scoped memory regions. All this supports predictable timing behaviours.

SCJ enforces particular programming patterns via simplified memory and concurrency models. SCJ programs can adopt one of three profiles, called levels, which support an increasing number of abstractions. In this paper, we focus on the intermediate level of SCJ programs (level 1), which enforces a structure where a safelet (the main program) defines a mission sequencer, which in turn provides a number of missions that are run in sequence as shown in Figure 1. This level is comparable in complexity to the Ravenscar profile for Ada. While adequate to a wide range of applications, it is amenable to formal reasoning.

An SCJ application is formed by a safelet, a mission sequencer, a number of missions, and periodic and aperiodic event handlers. A safelet instantiates a mission sequencer, and iteratively obtains a mission from the mission sequencer, executes it and waits for it to terminate. At level 1, each mission is formed by a collection of periodic and aperiodic event handlers that are run concurrently.

A safelet terminates when there are no missions left to be executed, and a mission terminates when one of its handlers requests termination. In the SCJ memory model, which is based on scoped memory regions, safelets, missions, and handlers each have associated memory regions, which are cleared at specific points of the program execution. This ensures predictable time properties.

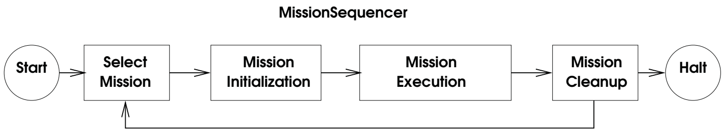


Fig. 1. SCJ programming model

Cavalcanti et al. [5] proposes a design technique for SCJ based on the *Circus* family of languages for refinement, which are state-rich process algebras for refinement that include aspects such as object-orientation and timing; it has been used to support verification of a number of different models such as Simulink/Stateflow diagrams [4,11] and SysML [13,9].

In [12], we describe the syntax and semantics of a version of *Circus* called *SCJ-Circus*, which includes the SCJ abstractions. In that work, we identify four patterns of *Circus* specifications used as a basis for the development of SCJ programs defined by *SCJ-Circus* models. We also describe the main phases of a refinement strategy that takes a *Circus* specification based on one of the patterns for a non-terminating cyclic design with one mission whose handlers are in lockstep. The strategy produces an *SCJ-Circus* program that can be directly converted into an SCJ program. Here, we show that a refinement strategy for the terminating versions of the patterns identified in [12] can be extended and composed to support the refinement of a wider variety of patterns, in particular, non-terminating and multi-mission patterns.

We first focus on the terminating cyclic in lockstep pattern, and detail the refinement strategy for it. We then show how this strategy can be extended to support the refinement of the non-terminating cyclic in lockstep pattern of [12] and a multi-mission pattern, in which the missions are provided sequentially and each mission follows the terminating cyclic in lockstep pattern. Besides considering refinement for a collection of patterns that is significantly larger than that in [12], we also provide a detailed account of the strategy, instead of just an overview. Whilst most of the laws used in our strategies have only syntactic provisos, a few require more complex provisos, such as deadlock freedom.

Section 2 introduces *Circus* and Section 3 discusses our patterns, focusing on the terminating cyclic in lockstep pattern. Section 4 details our refinement strategy for our target pattern. Section 5 and 6 extend the refinement strategy of Section 4 to cover non-terminating and multi-mission patterns. Finally, Section 7 concludes by reviewing our results and discussing future work.

2 *Circus*

In this section, we use the *Circus* process *ThreeEqual* in Figure 2, which models an SCJ level 1 application that contains two event handlers, to give an overview of the notation. It defines a program that takes integers as inputs, and outputs booleans indicating whether the last three inputs are equal or not.

$$
\begin{array}{l}
\mathbf{process}\ \mathit{ThreeEqual} \mathrel{\widehat{=}} \mathbf{begin}\\
\quad \mathit{InputHandler} \mathrel{\widehat{=}}\\
\qquad \mu\, X \bullet \left(\begin{array}{l} \left(\begin{array}{l} (\mathit{input}?x \longrightarrow \mathbf{Skip}) \blacktriangleleft \mathit{ID};\ \mathit{setBuffer}!x \longrightarrow \\ (\mathbf{wait}\, 0..\mathit{PTB});\ \mathit{checkRepeats} \longrightarrow \mathbf{Skip} \end{array}\right) \blacktriangleright \mathit{PD} \\ |||\ \mathbf{wait}\ P \end{array}\right);\ X\\
\quad \mathit{OutputHandler} \mathrel{\widehat{=}}\\
\qquad \mu\, X \bullet \left(\begin{array}{l} \mathit{checkRepeats} \longrightarrow \\ \qquad \left(\begin{array}{l} \mathit{getBuffer}?\mathit{buffer} \longrightarrow \mathbf{wait}\, 0..\mathit{ATB}; \\ \left(\begin{array}{l} \mathbf{if}\ \mathit{check}(\mathit{buffer}) = \mathbf{True} \longrightarrow \\ \quad (\mathit{output}!\mathit{true} \longrightarrow \mathbf{Skip}) \blacktriangleleft \mathit{OD} \\ [\!]\, \mathit{check}(\mathit{buffer}) = \mathbf{False} \longrightarrow \\ \quad (\mathit{output}!\mathit{false} \longrightarrow \mathbf{Skip}) \blacktriangleleft \mathit{OD} \\ \mathbf{fi} \end{array}\right) \end{array}\right) \blacktriangleright \mathit{AD};\ X \end{array}\right)\\
\quad \mathit{MArea} \mathrel{\widehat{=}} \mathbf{var}\ \mathit{buffer} : \mathrm{seq}\, \mathbb{N} \bullet \mathit{buffer} := \langle 0, 0, 0 \rangle;\ \mu\, X \bullet\\
\qquad \left(\begin{array}{l} \mathit{setBuffer}?x \longrightarrow \mathit{buffer} := (\mathit{tail}\ \mathit{buffer} \frown \langle x \rangle);\ X \\ \Box \mathit{getBuffer}!\mathit{buffer} \longrightarrow X \Box \mathit{stop} \longrightarrow \mathbf{Skip} \end{array}\right)\\
\quad \mathit{TEMission} =\\
\qquad \left(\begin{array}{l} \left(\begin{array}{l} \mathit{InputHandler} \\ [\![\{\} \mid \{\!| \mathit{stop}, \mathit{checkRepeats} |\!\} \mid \{\}]\!] \\ \mathit{OutputHandler} \end{array}\right) \setminus \{\!| \mathit{checkRepeats} |\!\} \\ [\![\{\} \mid \{\!| \mathit{setBuffer}, \mathit{getBuffer} |\!\} \mid \{\}]\!]\ \mathit{MArea} \end{array}\right) \setminus \{\!| \ldots |\!\}\\
\quad \mathit{TEMissionSequencer} \mathrel{\widehat{=}} \mathit{TEMission}\\
\quad \mathit{TESafelet} \mathrel{\widehat{=}} \mathit{TEMissionSequencer}\\
\quad \bullet\ \mathit{TESafelet}\\
\mathbf{end}
\end{array}
$$

Fig. 2. SCJ Level 1 example: ThreeEqual

The main modelling element of a *Circus* specification is a process (indicated by the keyword **process**). A basic process declares state components (identified by the keyword **state**), a number of auxiliary actions, and a main action (at the end prefixed by $\bullet$) that describes the overall behaviour of the process. Processes can also be combined using CSP operators to define other processes.

Processes communicate with each other and with the environment via channels. In the case of our example, the process *ThreeEqual* does not declare any state components; its interface is characterised by the channels *input* and *output*.

ThreeEqual declares six auxiliary actions: *InputHandler*, *OutputHandler*, *MArea*, *TEMission*, *TEMissionSequencer* and *TESafelet*. Actions are specified using a combination of Z [18] for data modelling and CSP [15] for behavioural descriptions. The main action is defined by a direct call to the action *TESafelet*.

In general, a safelet may have an initialisation, but in our example, its behaviour is just that of the action *TEMissionSequencer*. In general, a mission sequencer defines a sequence of missions, but here *TEMissionSequencer* defines just the mission *TEMission*. The action *MArea* represents a memory area that holds a buffer. It has a block ($\mathbf{var} \ldots \bullet \ldots$) that declares a local variable *buffer* of type $\mathrm{seq}\,\mathbb{N}$, and whose body is defined by a recursion ($\mu\, X \bullet \ldots$) that at each step offers a choice of reading a value on the (internal) channel *setBuffer*, storing it in *buffer* and recursing, or outputting the value of *buffer* on the channel *getBuffer* and recursing, or synchronising on *stop* and terminating. The action *Mission* composes in parallel ($[\![\ldots \mid \ldots \mid \ldots]\!]$) the two event handlers synchronising on *stop* and *checkRepeats*, with *checkRepeats* hidden ($\setminus$), and the action *MArea* synchronising on the channels *setBuffer* and *getBuffer*.

The action *InputHandler* represents a task with period P, that must get its input within ID time units, can take up to PTB time units to complete, but no more than PD. It is also defined by a recursion, where at each step two actions are started in interleaving. The first action must take at most PD time units as indicated by the end-by deadline operator $\blacktriangleright$. It reads an input with a deadline of ID time units as indicated by the start-by deadline operator $\blacktriangleleft$, appends the value to the end of the tail of *buffer*, waits (**wait**) between 0 and PTB time units, and then synchronises on *checkRepeats*, before terminating (**Skip**). The **wait** $0 \mathinner{..} PTB$ models a budget of time for the update of the buffer of PTB time units. We observe that data operations take no time, unless explicitly specified otherwise. The communication of *checkRepeats* triggers the *OutputHandler*. The second interleaved action in *InputHandler* waits for exactly P time units and guarantees that a new iteration of the recursion does not start before the end of the cycle, whose duration is P time units.

The action *OutputHandler* represents a task triggered by a synchronisation on the channel *checkRepeats*; the task can take up to ATB time units to complete, but must terminate in less than AD time units. It is defined by a recursion, where each step synchronises on *checkRepeats*, reads the value of the buffer on the channel *getBuffer*, waits for up to ATB time units, and checks whether the value read is in the set *check* or not. In the first case it outputs the value *true* on the channel *output* within OD time units, in the second case it outputs *false* on the same channel under the same time restriction. The whole step must terminate within AD time units as indicated by the deadline operator.

In general, a *Circus* specification consists of a sequence of paragraphs that define processes (as well as channels, constants, and other constructs that support the definition of processes). Processes are used to define the system and its components: state is encapsulated and interaction is via channels. Processes can be composed, via CSP operators, to define other processes. In *Circus Time*, wait and deadline operators can be used to define time restrictions. In *OhCircus* models, we can in addition define paragraphs that declare classes used to define types. More information about these languages can be found in [14,16,3]. In the sequel, we further explain the notation as needed.

3 Patterns

The cyclic executive pattern that has been identified in our previous work is shown in Figure 3. It requires that the application has a single mission with a fixed cycle, and all periodic and aperiodic event handlers execute at each cycle. This requirement radically simplifies the refinement strategy since it allows the

$$
\begin{array}{l}
P \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\, S \\
\quad \mathit{PHandler}_i \mathrel{\widehat=} \mu\, X \bullet (F_i \blacktriangleright \mathit{PD} \interleave \mathbf{wait}\, \mathit{PERIOD});\; X \,\Box\, t \longrightarrow \mathbf{Skip} \\
\quad \mathit{AHandler}_j \mathrel{\widehat=} \mu\, X \bullet (c_j \longrightarrow G_j) \blacktriangleright \mathit{AD};\; X \,\Box\, t \longrightarrow \mathbf{Skip} \\
\quad \mathit{MArea} \mathrel{\widehat=} \ldots \\
\quad \mathit{Termination} \mathrel{\widehat=} \mathit{rt} \longrightarrow \mu\, X \bullet (\mathit{rt} \longrightarrow X \,\Box\, t \longrightarrow \mathbf{Skip}) \\
\quad \mathit{Mission} \mathrel{\widehat=} \left(\begin{array}{l}
(\mathit{MArea} \parallel (\parallel i : I \bullet \mathit{PHandler}_i) \parallel (\parallel j : J \bullet \mathit{AHandler}_j)) \\
\quad [\![\alpha S \mid \{\!| t, \mathit{rt} |\!\} \mid \{\}]\!] \\
\mathit{Termination}
\end{array}\right) \\
\quad \mathit{MissionSequencer} \mathrel{\widehat=} \mathit{Mission} \\
\quad \mathit{Safelet} \mathrel{\widehat=} \mathit{MissionSequencer} \\
\quad \bullet\; \mathit{Safelet} \\
\mathbf{end}
\end{array}
$$

$$\mathbf{where}\ \mathit{PD} \leq \mathit{PERIOD} \wedge \mathit{AD} \leq \mathit{PERIOD}$$

Fig. 3. Pattern: cyclic in lockstep.

transformation of synchronous releases of aperiodic event handlers in the models into the asynchronous releases that occur in *SCJ-Circus*.

In this pattern, parallelism occurs between the actions that model the mission and the mission memory, and the termination management action, and within the mission action between the different event handlers. The mission memory action MArea declares the variables shared by the handlers and offers communications over $\mathit{get\_}$ and $\mathit{set\_}$ channels that support reading and writing to the shared variables. The termination management action accepts a synchronisation on a channel rt; this corresponds to a request from some handler to terminate. Afterwards, the action starts a recursion where at each step, it either accepts another communication on rt (corresponding to a further request to terminate) and recurses, or it synchronises on t and terminates. The communication on t has the effect of terminating the handlers of the mission.

A parallelism of actions in *Circus* needs to identify the partition of variables that each parallel action can modify to avoid race conditions. So far, these sets of variables have been empty. In the case of the $\mathit{Mission}$ action, all variables αS in scope can be modified by the parallelism of handlers, while the termination management action modifies no variables.

Periodic event handlers $\mathit{PHandler}_i$ are defined as recursive actions, where each step takes a fixed amount of time ($\mathbf{wait}\, \mathit{PERIOD}$) whilst executing (in interleave) some behaviour that takes at most PD time units (indicated by the $\blacktriangleright$ operator). The execution of an aperiodic event handler $\mathit{AHandler}_j$, on the other hand, is triggered by a synchronous event c_j, which may occur at any time during the cycle of the mission that lasts PERIOD time units, but must terminate within AD time units. In order to guarantee the cyclic behaviour, the pattern imposes timing restrictions over the behaviours of both periodic and aperiodic handlers, which guarantee that they terminate within the cycle of the mission.

The second pattern that we consider allows for an aperiodic event handler not to be called in a cycle. For this to be possible, the deadlines of the aperiodic event handler must be adapted so that they can be missed as long as it has not

$$AHandler_j \mathrel{\widehat=} \mu\, X \bullet (c_j \longrightarrow G_j) \blacktriangleright AD;\ X \mathrel{\Box} t \longrightarrow \mathbf{Skip} \mathrel{\Box} \mathbf{wait}\ PERIOD;\ X$$

$$\mathbf{where}\ AD \leq PERIOD$$

Fig. 4. Pattern: cyclic not in lockstep.

$$\begin{array}{l} AHandler_j \mathrel{\widehat=} \\ \quad \left(\begin{array}{l} \mathbf{var}\ pr : boolean \bullet pr := false; \\ \mu\, X \bullet \left(\begin{array}{l} c_j \longrightarrow pr := true;\ X \mathrel{\Box} \left(pr\right) \mathrel{\&} hr \longrightarrow pr := false;\ X \\ \Box t \longrightarrow \mathbf{Skip} \end{array}\right) \\ [\![\{\} \mid \{\!| hr, t |\!\} \mid \alpha(G_j)]\!] \\ \mu\, X \bullet (hr \longrightarrow G_j;\ X \mathrel{\Box} t \longrightarrow \mathbf{Skip}) \end{array}\right) \setminus \{\!| hr |\!\} \end{array}$$

Fig. 5. Pattern: non-cyclic.

started. This pattern differs from that in Figure 3 just in the characterisation of the aperiodic handlers, which is presented in Figure 4. It requires the release via the channel c_j and the associated handling action G_j to terminate within AD time units, or termination through the channel t. Furthermore, if release or termination does not take place within $PERIOD$ time units, the choice terminates and a new cycle of the recursion is started. The timing restrictions of the first pattern over the periodic event handlers also apply to the second pattern.

Finally, the third pattern imposes no restriction on the timing of the handlers, which means that the behaviour of a periodic handler can take longer than its period, and the behaviour of the aperiodic handler can be called multiple times even if its action is not yet completed. In this case, the pattern for aperiodic handler models is as shown in Figure 5. In SCJ, release requests are asynchronous. So, if requests for further releases occur before a handler is ready to execute again, one, and just one, pending release is recorded.

Accordingly, the actions of the form described in Figure 5 accumulate an asynchronous release. It is used to support accumulation of at most one asynchronous release and consists of two parallel actions. The second parallel action models the behaviour of the handler itself in the usual way, but it is triggered by the internal event hr. The actual triggering is managed by the first parallel action. It receives release requests through the channel c_j and records them in the local variable pr (pending release), or requests for termination on t. When there is a pending release (pr), then it can release the handler (hr), but it can never have more than one pending release. This is, no doubt, a very specific pattern, in spite of the absence of time restrictions. We emphasise that the starting point of our refinement strategy is not an abstract model, but an SCJ design. Such a design can be obtained using the refinement strategy in [5], for example.

The purpose of the refinement strategy that we propose here and detail in the next few sections is threefold. First, it guarantees that the design embedded by the patterns can indeed be realised in SCJ. Not every model that conforms to our patterns can be correctly implemented in SCJ with the suggested structure of missions and actions. For example, in the design model, there may be a possibility that an aperiodic handler is not released in a particular cycle. Such a model

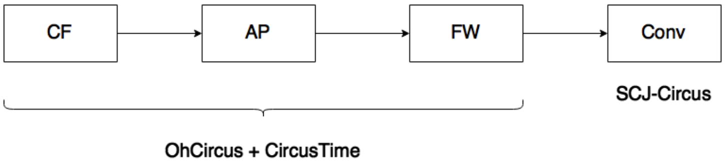


Fig. 6. Phases of the refinement strategy

$\mathbf{safelet}\,\mathit{Safelet} \mathrel{\widehat=} \mathbf{begin} \ldots \mathbf{end}$
$\mathbf{sequencer}\,\mathit{Sequencer} \mathrel{\widehat=} \mathbf{begin} \ldots \mathbf{end}$
$\mathbf{mission}\,\mathit{Mission} \mathrel{\widehat=} \mathbf{begin} \ldots \mathbf{end}$
$\mathbf{periodic\ handler}\,\mathit{PHandler}_i \mathrel{\widehat=} \mathbf{begin} \ldots \mathbf{handleAsyncEvent} \mathrel{\widehat=} F_i \blacktriangleright PD\ \mathbf{end}$
$\mathbf{aperiodic\ handler}\,\mathit{AHandler}_j \mathrel{\widehat=} \mathbf{begin} \ldots \mathbf{handleAsyncEvent} \mathrel{\widehat=} G_j \blacktriangleright AD\ \mathbf{end}$

Fig. 7. Target

cannot be realised as a cyclic execution in lockstep. Its rendering in SCJ may lead to visible inputs and outputs not allowed in the model.

Second, by deriving via refinement an SCJ-Circus model, we enable automatic translation to SCJ code via trivial transformations whose soundness is not a concern. Finally, we obtain a model whose abstractions are in direct correspondence with those of the SCJ paradigm. In this model, reasoning about use of the memory model, for example, is much simpler.

4 Refinement strategy

In this section, we present the refinement strategy for the terminating cyclic lockstep pattern. It consists of the same phases as the refinement strategy in [12] shown in Figure 6. The target is an *SCJ-Circus* program of the form defined in Figure 7. Each periodic action in the starting design model has a corresponding periodic event handler paragraph in the target specification, where the **handleAsyncEvent** method is defined as $F_i \blacktriangleright PD$. Similarly, each aperiodic, mission, mission sequencer and safelet action has a corresponding paragraph.

The four phases of our refinement strategy are as follows: (CF) introduction of SCJ control flow, (AP) introduction of application processes, (FW) introduction of framework processes, (Conv) conversion to *SCJ-Circus*. CF makes the control flow of the SCJ paradigm, which is implicitly defined in the patterns via sequential and parallel compositions, explicitly captured by channel synchronisations. For example, we introduce, a channel *activate_handlers* that models the synchronised start of the handlers in parallel. AP separates application-specific behaviours (for example, handler behaviours) from behaviours such as starting a mission, which are implemented by an SCJ runtime environment (framework). FW takes the incomplete model of framework behaviour, representing a slice of the SCJ framework actually used by the application, and replaces it by the full-fledged framework model. Finally, Conv refines the specification into a sequence of *SCJ-Circus* paragraphs. As illustrated in Figure 6, the first three phases act only on constructs of the time and object-oriented languages, *Circus Time* and *OhCircus*, whilst the last phase manipulates *SCJ-Circus* specifications, which complement those two notations with SCJ specific constructs.

4.1 (CF) Introduction of SCJ control flow

This phase extracts each of the SCJ abstractions from the starting design model into parallel actions. It derives, from a design like that in Figure 3, a process structured as shown in Figure 8. Its main action is the parallel composition of actions corresponding to specific SCJ abstractions. The order of execution imposed by the original specification is maintained through the use of communication channels such as *start_mission* and *start_sequencer*.

Figure 9 presents the steps necessary to reach its target. The laws named there can be found in [1]. Due to space restrictions, only some are presented and explained here. Overall, this phase identifies the actions of the starting process that model specific abstractions and applies specialised laws to parallelise the safelet, mission sequencer and mission actions. Next, handler laws replace synchronous communications between handlers with asynchronous communications, and separate the handlers from the mission action. Finally, *Circus* laws are used to merge parallel actions associated with mission execution.

The law call-intro is used to split an action $F(A)$ into the parallel composition of two actions, one of which executes the behaviour of the subaction A of $F(A)$. To retain the control flow of $F(A)$, internal channels cs and ce are used to synchronise the parallel actions. In $F(A)$, the action A is replaced with a call to the parallel action using the internal channels.

Law [call-intro]

$$F(A) \sqsubseteq \left(\begin{array}{l} F(cs \longrightarrow ce \longrightarrow \mathbf{Skip}) \\ [\![\overline{wrtV(A)} \mid \{\!|\, cs, ce \,|\!\} \mid wrtV(A)]\!] \\ cs \longrightarrow A;\ ce \longrightarrow \mathbf{Skip} \end{array} \right) \setminus \{\!|\, cs, ce \,|\!\}$$

provided

- $\{\!|\, cs, ce \,|\!\} \cap usedC(A) = \varnothing$
- $wrtV(A) \cap usedV(F(\mathbf{Skip})) = \varnothing$
- $wrtV(F(\mathbf{Skip}) \cap usedV(A) = \varnothing$

This law is proved by structural induction over the structure of the action F using distribution and step laws such as the ones found in [14]. The provisos guarantee that the internal channels are fresh and that the state is appropriately partitioned to avoid racing conditions in the parallelism. We use $usedC(A)$ to refer to the set of channels used in an action A, and $usedV(A)$ and $wrtV(A)$ to refer to the variables used and modified by A. The law call-intro is applied to parallelise the safelet, mission sequencer and mission.

$$
\begin{array}{l}
\mathit{CF\_P} \mathrel{\widehat=} \mathbf{begin}\\
\mathbf{state}\ S\\
\mathit{PHandler}_i \mathrel{\widehat=} \mu\, X \bullet (F_i \blacktriangleright \mathit{PD} \mathbin{|||} \mathbf{wait}\ \mathit{PERIOD});\ X \mathbin{\Box} t \longrightarrow \mathbf{Skip}\\
\mathit{AHandler}_j \mathrel{\widehat=} \mu\, X \bullet (c_j \longrightarrow G_j) \blacktriangleright \mathit{AD};\ X \mathbin{\Box} t \longrightarrow \mathbf{Skip}\\
\mathit{MArea} \mathrel{\widehat=} \ldots\\
\mathit{Termination} \mathrel{\widehat=} \mathit{rt} \longrightarrow \mu\, X \bullet (\mathit{rt} \longrightarrow X \mathbin{\Box} t \longrightarrow \mathbf{Skip})\\
\mathit{CF\_Mission} \mathrel{\widehat=} \mathit{start\_mission} \longrightarrow\\
\quad \left(\begin{array}{l}
\mathit{MArea} \parallel \mathit{Termination} \parallel\\
\left(\begin{array}{l}
\parallel i : I \bullet \mathit{SH}_i \longrightarrow \mathit{register}.i \longrightarrow \mathit{start\_peh}.i \longrightarrow \mathit{activate\_handlers} \longrightarrow\\
\quad \mathit{done\_handler}.i \longrightarrow \mathbf{Skip}\\
\parallel\\
\parallel j : J \bullet \mathit{SH}_j \longrightarrow \mathit{register}.j \longrightarrow \mathit{start\_aeh}.j \longrightarrow \mathit{activate\_handlers} \longrightarrow\\
\mathit{quaddone\_handler}.j \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}\right);\\
\quad \mathit{done\_mission} \longrightarrow \mathbf{Skip}\\
\mathit{Safelet} \mathrel{\widehat=}\\
\quad \left(\begin{array}{l}
\parallel i : I \bullet \mathit{SH}_i \longrightarrow \mathit{start\_peh}.i \longrightarrow \mathit{activate\_handlers} \longrightarrow \mathit{PHandler}_i;\\
\quad \mathit{done\_handler}.i \longrightarrow \mathbf{Skip}\\
\mathbin{|||} j : J \bullet \mathit{SH}_j \longrightarrow \mathit{start\_aeh}.j \longrightarrow \mathit{activate\_handlers} \longrightarrow\\
\quad (\mathit{AHandler}_j \llbracket \{\ldots\} \mid \{\!|\, c_{ji} \,|\!\} \mid \{\} \rrbracket\ \mathit{Buffer}_j) \setminus \{\!|\, c_{ji} \,|\!\};\\
\quad \mathit{done\_handler}.j \longrightarrow \mathbf{Skip}\\
\parallel \mathit{CF\_Mission}\\
\parallel \mathit{start\_sequencer} \longrightarrow \mathit{start\_mission} \longrightarrow \mathit{done\_mission} \longrightarrow\\
\quad \mathit{done\_sequencer} \longrightarrow \mathbf{Skip}\\
\parallel \mathit{start\_sequencer} \longrightarrow \mathit{done\_sequencer} \longrightarrow \mathbf{Skip}
\end{array}\right)\\
\bullet\ \mathit{Safelet}\\
\mathbf{end}
\end{array}
$$

Fig. 8. Refinement strategy (CF) – Target

1. Apply Law call-intro to the action *Safelet* with channels *cs* and *ce* replaced by *start_sequencer* and *done_sequencer*;
2. Apply Law call-intro to the action *MissionSequencer* with channels *cs* and *ce* replaced by *start_mission* and *done_mission*;
3. Apply Law copy-rule to the action *Mission* in *MissionSequencer*;
4. For each aperiodic action $\mathit{AHandler}_j$ in the parallelism of handler actions use associativity and commutativity laws to obtain a parallelism between $\mathit{AHandler}_j$ and another parallelism with all other handlers, and apply Law sync-async-conv;
5. Apply Laws prefix-par-dist [14] and par-prefix-dist to the action *Mission* to distribute the communications on *start_mission* and *done_mission* over all parallel actions;
6. Apply Law handler-extract to each parallel action except *MArea* and *Termination*;
7. Apply step laws of [14] to merge the left hand side actions of the parallelisms introduced in the previous step.

Fig. 9. Refinement strategy: (CF) Introduction of SCJ control flow

For the treatment of event handlers, we first introduce asynchronous communication between the event handlers using the Law sync-async-conv (see step 4 of Figure 9). This law, which we omit here, replaces the synchronous communication between two actions by an asynchronous communication based on a buffer.

$$
\begin{array}{l}
\mathit{Handler}_i\_\mathit{app} \mathrel{\widehat{=}} \ldots \\
\mathit{Mission}\_\mathit{app} \mathrel{\widehat{=}} \ldots \\
\mathit{MissionSequencer}\_\mathit{app} \mathrel{\widehat{=}} \ldots \\
\mathit{Safelet}\_\mathit{app} \mathrel{\widehat{=}} \ldots \\
AP\_P \mathrel{\widehat{=}} AP\_P\_FW \parallel \left(\begin{array}{l} \mathit{Safelet}\_\mathit{app} \mid\mid\mid \mathit{MissionSequencer}\_\mathit{app} \mid\mid\mid \mathit{Mission}\_\mathit{app} \parallel \\ (\mid\mid\mid i : I \cup J \bullet \mathit{Handler}_i\_\mathit{app}) \end{array} \right)
\end{array}
$$

Fig. 10. Refinement strategy: target of phase **AP**

Next, parallelism distribution laws are used in step 5 to expand the parallelism between handlers (previously internal to the action *Mission*) to a top level parallelism as shown in Figure 8. In the next step, a simple law, handler-extract omitted here, is used wrap the handler actions with synchronisations on new internal channels *SH*, *register*, *start_peh*, *start_aeh*, *activate_handlers* and *done_handler* to represent the interactions corresponding to the initialisation of a mission, including creation (*SH*), registration (*register*), starting (*start_peh* and *start_aeh*) and activation (*activate_handlers*) of handlers, and to the termination of a mission, including the cleaning of handlers (*done_handler*). All these synchronisations are orchestrated by a new parallel action that models the mission execution cycle. Its repeated occurrence for each handler is eliminated in favour of a single parallel action named *CF_Mission* in Figure 8.

4.2 (AP) Introduction of application processes

The starting point of this phase is the target of the previous phase in Figure 8, and its target is shown in Figure 10: it defines a number of application processes, and refines the process *CF_P* into the parallel composition of the interleaved application processes and a modified version of the original process, where application-specific behaviours have been replaced by calls to actions of the application processes via *Call* and *Ret* channels that model method calls.

The steps of this phase are shown in Figure 11. Overall, we use the process obtained in phase CF to identify the behaviours that are application specific and construct application processes. Next, each action modelling an SCJ abstraction is split into two parallel actions: one containing application-specific behaviours, and the other containing the interactions introduced during CF to model the SCJ control flow. In this control action, the application-specific behaviour is replaced by calls via appropriate channels. This is achieved by specialised laws handler-split, mission-split, sequencer-split and safelet-split, for each of the different SCJ constructs. Finally the initial basic process is split into a parallelism of processes. Since, following the application of the specialised split laws, the main action of the basic process is a parallelism, this is a simple application of the definition of process parallelism in *Circus*.

Due to space restrictions, we present just the Law handler-split. The others are similar and simpler. For handlers, the new parallel actions communicate through channels that model a call to the *handleAsyncEvent* method. Accordingly, this

1. Apply Law handler-split to each handler action with channels $haeC$ and $haeR$ replaced by $handleAsyncEventCall$ and $handleAsyncEventRet$;
2. Apply Law mission-split to the action that models the mission;
3. Apply Law seq-interleave [1] to turn the interleaving on the left hand side of the parallel action introduced in step 2 into a sequential composition;
4. Apply Law rec-interleave [1] to turn the interleaving on the right hand side of the parallel action in step 2 into a recursion;
5. Apply Law sequencer-split [1] to the action that models the sequencer;
6. Apply Law safelet-split [1] to the action that models the safelet;
7. Apply the definition of parallel processes [14] from right to left to replace the basic process, whose main action is parallel, with a parallelism of application processes and the remains of the original process.

Fig. 11. Refinement strategy: (AP) Introduction of application processes

law takes an action modelling a handler, and splits it by distributing application aspects such as constructor channels SH and release behaviour F to one side, and framework behaviours such as start and end channels (sh and dh) to the other side. This law is easily proved by the application of parallelism step laws [14].

Law [handler-split]

$$SH.n \longrightarrow sh.n \longrightarrow \mu X \bullet (F;\ X);\ dh.n \longrightarrow \mathbf{Skip}$$

$$\sqsubseteq$$

$$\left(\begin{array}{l} SH.n \longrightarrow sh.n \longrightarrow \mu X \bullet (haeC \longrightarrow F;\ haeR \longrightarrow X);\ dh.n \longrightarrow \mathbf{Skip} \\ \llbracket wrtV(F) \mid \{\!| sh.n, dh.n, haeC, haeR |\!\} \mid \{\} \rrbracket \\ sh.n \longrightarrow \mu X \bullet (haeC \longrightarrow haeR \longrightarrow X);\ dh.n \longrightarrow \mathbf{Skip} \end{array}\right)$$
$$\setminus \{\!| haeC, haeR |\!\}$$

provided $\{\!| sh, SH, dh |\!\} \cap usedC(F) = \varnothing$.

In step 3, the strategy applies a law to transform the interleaving of the instantiation and registration of all handlers (on the application side) into a sequence. This is possible because all the parallel actions that synchronise on the interleaved events do so in interleaving (avoiding deadlock) and these events are internal. Step 4 transforms the interleaving on the framework side of the mission into a recursive action that at each step allows the registration of a handler, and once all handlers have been registered, executes them in interleaving.

At the end of this phase, the application processes are completed, but the remaining process AP_P_FW does not quite specify the SCJ runtime environment. This process is the focus of the next phase.

4.3 (FW) Introduction of framework processes

This phase applies to the part of the model that remains after the application processes are extracted. It consists of a process AP_P_FW containing portions

1. Apply Law safelet-fw-cl to the action *Safelet* of CF_P_FW;
2. Apply Law sequencer-fw-cl to the action *Sequencer* of CF_P_FW;
3. Apply Law mission-fw-cl to the action *Mission* of CF_P_FW;
4. Apply Law periodic-handler-fw-cl to the actions $PHandler_i$ of CF_P_FW;
5. Apply Law aperiodic-handler-fw-cl to the action $AHandler_j$ of CF_P_FW;
6. Apply the definition of parallel processes [14] from right to left to replace the process CF_P_FW, with a parallelism of framework processes.

Fig. 12. Refinement strategy: (FW) Introduction of framework processes

$$FW\_P \mathrel{\widehat=} \left(\begin{array}{l} \left(\begin{array}{l} SafeletFW \parallel SequencerFW \parallel MissionFW(mission) \parallel \\ (\interleave i : I \cup J \bullet HandlerFW(handler_i)) \end{array} \right) \\ \parallel \\ \left(\begin{array}{l} Safelet\_app \interleave MissionSequencer\_app \interleave Mission\_app \parallel \\ (\interleave i : I \cup J \bullet Handler_i\_app) \end{array} \right) \end{array} \right)$$

Fig. 13. Refinement strategy: target of phase **FW**

of the framework that are explicitly used in the design. The result is a new process that defines the complete framework behaviours. For instance, our running example never asks a mission for the sequencer that oversees its execution. This is, however, a service provided by the framework. We can introduce the richer description of the framework because the application process is guaranteed not to request the additional behaviour. The steps of this phase are shown in Figure 12.

The process resulting from the application of this phase is shown in Figure 13. It is the parallel composition of the application processes introduced in the previous phase and the framework processes that model the SCJ API.

The main laws used in this phase are specialised to the cyclic in lockstep pattern as indicated by the suffix -cl. The single non-application process AP_P_FW obtained in the previous phase is the same for all applications that follow our target pattern. This is because the pattern is very restrictive with respect to the execution of missions and handlers, and most of the framework specific behaviours are introduced by the laws in the previous steps. For this reason, the specialised laws can be used to introduce the full blown framework processes relying solely on syntactic conditions over the application processes. This is done to each abstraction in steps 1–5. These framework processes are specified in [10].

At the final step 6, the process whose main action is the parallel composition of the actions completed by the previous steps is split into a parallelism of framework processes. The result is a parallelism of processes as shown in Figure 13.

4.4 (Conv) Conversion to *SCJ-Circus*

This phase rearranges the parallel processes shown in Figure 13 by pairing framework and application processes according to the abstraction they model, and introducing new process paragraphs that isolate these pairs. For instance,

handler $S\_Handler_i \mathrel{\widehat=} \ldots$
mission $S\_Mission \mathrel{\widehat=} \ldots$
sequencer $S\_MissionSequencer \mathrel{\widehat=} \ldots$
safelet $S\_Safelet \mathrel{\widehat=} \ldots$

Fig. 14. Refinement strategy: target of phase **Conv**

1. Systematically apply Law par-par-dist to rearrange the parallelism in Figure 13, until it is structured as a parallelism of pairs of processes;
2. For each pair of processes, apply the copy rule from right to left to introduce the corresponding action paragraph and replace the process by a call;
3. For each newly introduced action, apply the definition of the appropriate SCJ abstraction from right to left.

Fig. 15. Refinement strategy: (Conv) Conversion to *SCJ-Circus*

$SafeletFW$ is paired with $Safelet\_app$, and extracted into a process $Safelet$. Next, the semantics of *SCJ-Circus* is used to convert the newly introduced processes into the corresponding *SCJ-Circus* paragraphs. For example, the process $Safelet$ is converted into a paragraph identified by the keyword **safelet**.

The target of this phase is a specification formed by *SCJ-Circus* paragraphs as shown in Figure 14. Each action that models an SCJ abstraction in the original design is modelled by an *SCJ-Circus* paragraph. These paragraphs overtly specify only the application specific behaviours, leaving the framework aspects implicit.

The steps for this phase are shown in Figure 15. The first step extracts pairs of application and framework processes two by two using the Law par-par-dist below. This is carried out exhaustively until there are no more pairs to extract.

Law [par-par-dist]

$$(A \llbracket s_A \mid a_1 \mid s_B \rrbracket B) \llbracket s_A \cup s_B \mid a_2 \cup b \mid s_C \cup s_D \rrbracket (C \llbracket s_C \mid c \mid s_D \rrbracket D)$$
$$=$$
$$(A \llbracket s_A \mid a_2 \mid s_C \rrbracket C) \llbracket s_A \cup s_C \mid a_1 \cup c \mid s_B \cup s_D \rrbracket (B \llbracket s_B \mid b \mid s_D \rrbracket D)$$

provided

- $usedC(A) \cap usedC(B) \subseteq a_1 \land a_1 \cap usedC(C, D) = \varnothing$
- $usedC(C) \cap usedC(D) \subseteq c \land c \cap usedC(A, B) = \varnothing$
- $usedC(A) \cap usedC(C) \subseteq a_2 \land a_2 \cap usedC(B, D) = \varnothing$
- $usedC(B) \cap usedC(D) \subseteq b \land b \cap usedC(A, C) = \varnothing$

This law relies on the strict partition of the communication network between the four parallel processes. It uses the fact that the channels used by the processes C and D, which are matched to application processes in our strategy, to communicate with each other are not used by A and B, and, conversely, that the channels used by A and B (matched to the framework processes in our strategy) to communicate with each other are not used by the application processes.

Next, each pair of application and framework processes is used to define a new process using the reverse of the copy-rule, and the semantics of *SCJ-Circus* is used to transform these newly defined processes into *SCJ-Circus* paragraphs.

5 Non-terminating pattern

Figure 16 shows the *Mission* action of the non-terminating cyclic in lockstep pattern. The main difference from that in Figure 3 is the missing Termination action. The target of our refinement strategy is the same: an *SCJ-Circus* program in the form described in Figure 7.

$$Mission \mathrel{\widehat=} (MArea \parallel (\parallel i : I \bullet PHandler_i) \parallel (\parallel j : Jn \bullet AHandler_j)$$

Fig. 16. Non-terminating cyclic in lockstep pattern

The refinement strategy described in Section 4 cannot be applied to models that follow the pattern in Figure 16 because it expects the *Mission* action to have an extra parallelism: see step 6 of CF and step 3 of FW. Instead of modifying the mission specific laws to introduce the mechanisms of termination, it is possible to extend the refinement strategy in the Section 4 by introducing this parallel action as a first step using the Law termination-intro, omitted here, before applying it.

This law takes a *Circus* action A of the form $\mu X \bullet F;\ X$ and two channels t and rt, and refines A into a parallelism between $\mu X \bullet F;\ X \,\Box\, t \longrightarrow \mathbf{Skip}$, and an action that waits for a termination request on a channel rt and then behaves as a recursive action that either accepts an event on the channel rt and recurses, or accepts an event on the channel t and terminates. The parallelism synchronises on both t and rt, which are made internal via the hiding operator. This law relies on the fact that A does not terminate, and does not use rt or t.

It may seem inefficient to complicate the model, but we note that the refinement steps of the whole refinement strategy are mostly automatic. Moreover, the phase FW is already about completing the framework model to reflect the SCJ paradigm. The termination protocol is part of the framework model already.

6 Multiple terminating missions

For an application with multiple missions in sequence, the pattern only differs in the specification of the action *MissionSequencer*, which is defined as the sequential composition of a number of missions $M_1;\ M_2;\ \ldots;\ M_n$. In this case, it is possible to modify the existing refinement strategy at very specific points to cater for a sequence of missions.

Step 2 of CF needs to be replaced with an iteration that, for each mission M_i, applies the Law call-intro to *MissionSequencer* with A instantiated as M_i and the channels cs and ce replaced by $start\_mission.M_i\_ID$ and $done\_mission.M_i\_ID$. With that, the mission-sequencer action is refined to a sequence of pairs of synchronisations on $start\_mission$ and $done\_mission$, in parallel with actions that call the mission actions. We have one parallel action for each mission, with the call wrapped by the $start\_mission$ and $done\_mission$ events. This is similar to the result obtained for the first pattern: the only difference is that, in this case, we have several parallel calls to missions.

Next, for each mission, the modified strategy applies the remaining steps described originally, including those of the following phases. We have to take into account, however, that the steps 5 and 6 of AP and 1 and 2 of FW only need to be applied in the first iteration. These steps are related to the application and framework processes for the safelet and the mission sequencer, and need to be carried out just once. Moreover, step 5 of AP needs a slightly different refinement law, which introduces a particular pattern for the implementation of the `getNextMission` tailored for multiple missions in sequence.

7 Conclusions

In this paper, we detail a refinement strategy for SCJ specifications. We describe each step necessary, and present some of the specialised laws required. This strategy differs from the refinement strategy for SCJ in [5] in that the latter takes an abstract model and refines it into a concrete program using specification patterns based on SCJ, but not its API. The strategy we present here refines a concrete SCJ-based model into a program that makes full use of the standard SCJ library to implement control aspects that are specific to SCJ. In that sense, our refinement strategy is similar to compilation, except that the target *SCJ-Circus* programs include library calls that are not present in the starting model. Moreover, some of the applications of refinement laws in the strategy generate proof obligations. Theorem proving is required when applying the strategy.

Despite that, since the *Circus* models used as a starting point for our strategy already embed an SCJ design, the level of automation achievable in applying the strategy is much higher than in [5]. For the particular pattern that we target here, most of the laws used have only syntactic provisos.

Our refinement strategy, possibly in combination with the one in [5], complements other verification techniques for SCJ. The work in [8] proposes an annotation-free technique for the verification of memory safety of SCJ programs based on a translation to a notation similar to *SCJ-Circus*. Also, [6] extends the widely used Java Modelling Language [2] with timing annotations to support worst case execution time analysis of SCJ programs.

It is worth mentioning that the pattern on which we focus here is fairly common in safety critical systems. For instance, the refinement strategy for control law diagrams proposed in [4] targets Ada implementations that follow a similar pattern, and it may be possible to adapt both refinement strategies to support the verification of SCJ implementations of control law diagrams.

As future work, we plan to refactor our strategy by extracting a refinement strategy that targets missions. With this structure, our strategy can be more easily generalised. We plan to specify strategies that target common patterns of mission combination, as well as missions following different patterns. Finally, we plan to implement our strategies in a theorem prover such as Isabelle/HOL, and apply them to existing examples such as a collision detection system [19].

Acknowledgements This work is funded by the EPSRC grant EP/H017461/1. No new primary data was created during this study.

References

1. A. Miyazawa and A. Cavalcanti. Report on refinement strategies for Safety-Critical Java, 2015. `http://www-users.cs.york.ac.uk/~alvarohm/report2015b.pdf`.
2. L. Burdy et al. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, June 2005.
3. A. Cavalcanti, A. Sampaio, and J. Woodcock. Unifying classes and processes. *Software & Systems Modeling*, 4(3):277–296, 2005.
4. A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via *Circus*. *Formal Aspects of Computing*, 23(4):465 – 512, 2011.
5. A. L. C. Cavalcanti et al.Safety-Critical Java programs from *Circus* models. *Real-Time Systems*, 49(5):614–667, 2013.
6. G. Haddad et al. The design of SafeJML, a specification language for SCJ with support for WCET specification. In JTRES '10, pages 155–163. ACM, 2010.
7. D. Locke et al.Safety-Critical Java technology specification. Technical report.
8. C. Marriott and A. L. C. Cavalcanti. SCJ: Memory-safety checking without annotations. In *FM*, volume 8442 of *LNCS*, pages 465–480. Springer, 2014.
9. A. Miyazawa and A. Cavalcanti. Formal refinement in SysML. In *iFM 2014*, volume 8739 of *LNCS*, pages 155–170. Springer, 2014.
10. A. Miyazawa and A. Cavalcanti. Refinement of *Circus* models into *SCJ-Circus*, 2015. `http://www-users.cs.york.ac.uk/~alvarohm/report2015a.pdf`.
11. A. Miyazawa and A. L. C. Cavalcanti. Refinement-oriented models of Stateflow charts. *Science of Computer Programming*, 77(10-11):1151–1177, 2012.
12. A. Miyazawa and A. L. C. Cavalcanti. *SCJ-Circus*: a refinement-oriented formal notation for Safety-Critical Java. In *Refinement Workshop*, 2015.
13. A. Miyazawa, L. Lima, and A. Cavalcanti. Formal models of SysML blocks. In *ICFEM 2013*, volume 8144 of *LNCS*, pages 249–264. Springer, 2013.
14. M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs Using Circus*. PhD thesis, University of York, 2006.
15. A. W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2011.
16. K. Wei, J. C. P. Woodcock, and A. L. C. Cavalcanti. *Circus Time* with Reactive Designs. In *UTP 2012*, volume 7681 of *LNCS*, pages 68–87. Springer, 2012.
17. A. Wellings. *Concurrent and Real-Time Programming in Java.* John Wiley & Sons, 2004.
18. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.
19. F. Zeyda, A. Cavalcanti, A. Wellings, J. Woodcock, and K. Wei. Refinement of the Parallel CDx. Technical report, University of York, 2012.

Automatic generation of test cases and test purposes from natural language

Sidney Nogueira[1,2], Hugo L. S. Araujo[1], Renata B. S. Araujo[1], Juliano Iyoda[1], and Augusto Sampaio[1]

[1] Centro de Informática, Universidade Federal de Pernambuco
[2] DEINFO, Universidade Federal Rural de Pernambuco

Abstract. Use cases are widely used for requirements description in the software engineering practice. As a use case event flow is often written in natural language, it lacks tools for automatic analysis or processing. In this paper, we extend previous work that proposes an automatic strategy for generating test cases from use cases written in a Controlled Natural Language (CNL), which is a subset of English that can be processed and translated into a formal representation. Here we propose a state-based CNL for describing use cases. We translate state-based use case descriptions into CSP processes from which test cases can be automatically generated. In addition, we show how a similar notation can be used to specify test selection via the definition of state-based test purposes, which are also translated into CSP processes. Test generation and selection are mechanised by running refinement checking verifications using the CSP processes for use cases and test purposes. All the steps of the strategy are integrated into a tool that provides a GUI for authoring use cases and test purposes described in the proposed CNL, so the formal CSP notation is totally hidden from the test designer. We illustrate our tool and techniques with a running example.

1 Introduction

Model-based testing (MBT) uses a model to describe the system under test in order to generate test cases automatically. There are many different notations used to create test models. For instance, Finite State Machines (FSM) and Labelled Transition Systems (LTS) are commonly adopted as input for test generation tools [12, 25]. Therefore, the adoption of MBT normally requires the users to be familiar with a formal specification language.

Unfortunately, in some contexts, working with modelling formalisms might be an impediment for the effective use of MBT. Thus, a more light-weight notation supported by an interactive tool might help to reduce the gap between informal specifications adopted in the initial phases of software development processes and a formal representation required by MBT.

Use cases are widely employed in the software engineering practice. It describes interactions between the user and the system in terms of event flows, typically written in natural language. Use cases are easier to learn in comparison with the formal notations used in MBT. However, as natural language is hard to process automatically, use cases lack tool support for automatic analysis and processing. Some efforts in this direction generate test cases from requirements written in natural language [3, 4, 19, 21]. In [3, 19] the authors do not impose any grammar restrictions on the language. This tends to demand more user intervention to generate test cases, unless there is a substantial knowledge base for a specific domain. The works reported in [4, 21] define standardised notations and provide automated strategies to generate test cases.

In previous work [15] we devised a strategy for the automatic generation of test cases from use cases written in a Controlled Natural Language (CNL), which is a subset of English that can be processed and translated into a formal language, and particularly CSP [11]. The main limitation of this strategy, however, is that states, inputs and outputs are expressed directly using (a variation of) the CSP notation, which, in general, is not accessible for test designers.

Here we propose a state-based CNL that embodies state and state operations, as well as inputs and outputs. Despite being a natural language, it requires from the use case designer familiarity with notions of variables, inputs and outputs. We translate state-based CNL use cases into a CSP representation that includes a memory model to record state information. From such a model, test cases and test data are automatically generated. A similar notation has proved convenient to express test purposes as a mechanism for test selection; this uniform use of our state-based CNL is a distinguishing feature of our approach with respect to related work. Furthermore, test generation is normally achieved by designing and running explicit algorithms that traverse a test model. By using CSP to represent both control behaviour and state information, we are able to generate test cases and test data as counterexamples of refinement verifications using a tool such as FDR [10]. Selection can be based both on the occurrence of events as well as on the system state. Our overall approach is illustrated by a case study, presented as a running example.

The next section gives an overview of CSP and of our previous strategy to test case generation. Section 3 introduces the proposed state-based controlled natural language for use case. The following section addresses the translation of state-based CNL descriptions into CSP. Then, Section 5 introduces the CNL notation for test purpose specification. In Section 6 we present tool support. In the final section we summarise our results and discuss related and future work.

2 Test Case Generation from CSP Models

This section overviews our previous strategy [15] for test case generation from use cases written in natural language, as well as the CSP notation. Figure 1 gives an overview of the TaRGeT Tool [9] that mechanises the strategy. The main inputs are use case documents that specify the software features to be tested, and test purposes written in CSP that define subsets of test scenarios to be generated. Feature descriptions are then translated to CSP test models that capture use cases behaviour. Test models and test purposes are inputs for our test generation approach, which uses FDR [10] to run verifications that yield the desired set of abstract test cases represented as CSP traces (sequences of events). Finally, CNL test cases are obtained from the abstract CSP test suite.

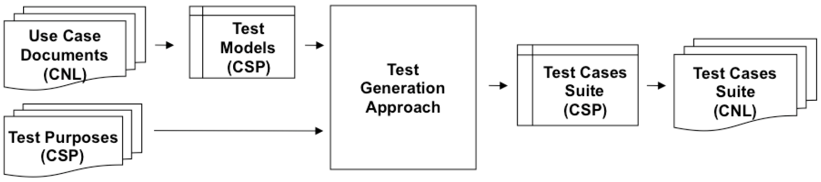


Fig. 1. Test generation approach in the TaRGeT tool

Use Case Documents Use case documents follow a template that allows the specification of several features; each feature has one or more use cases. Sentences are used to describe user actions and the respective system responses. The CNL is restricted to control aspects. Input, output, guards and variable update can be specified, but they are described in a formal notation (a variant of CSP).

CSP Test Models For the purpose of test case generation, the use case documents are translated into a formal representation in the CSP process algebra [17, 18]. The main element in CSP is a process, which models a component or an entire system. Processes communicate (atomic and instantaneous) events that represent visible actions. The alphabet α_P denotes the set of events a CSP process P can communicate. Events are communicated as soon as there is an agreement between processes and the environment. $STOP$ and $SKIP$ are two primitive CSP process. The former represents a deadlocked process that does not communicate any events. The latter represents a process that terminates successfully (communicates the ✓ event) and deadlocks.

Basic CSP operators as prefix, external choice, and sequential composition are suitable to model the control flow of use cases. The CSP prefix operator $P = ev \to Q$ specifies that event ev is communicated by P, which then behaves as the process Q. A channel is an important abstraction in CSP for a set of events with a common prefix. Let T be a type and c a channel that can communicate a value of type T (declared as $c : T$). Such a channel declaration represents the set of events $\{c.x \mid x \in T\}$. The external choice operator $P = Q \Box R$ indicates that the process P can behave as Q or R; the choice is made by the environment. The sequential composition $P;\ Q$ behaves initially like P; if it terminates successfully, then the control passes to Q.

Parallel composition, hiding and interruption are other CSP operators used in the model for a state-based use case. The process $P \mid[\, X \,]\mid Q$ stands for the generalised parallel composition of the processes P and Q with synchronisation set X. This expression states that P and Q must synchronise on events that belong to X. Each process can progress independently for events that are not in X. This composition terminates successfully if, and only if, the left- and the right-hand side processes do terminate. Likewise, consider the process $P \backslash X$ that communicates all its events, except the events that belong to X, which become internal (invisible): \ stands for the hiding operator. We also introduce the process $P \ \triangle \ Q$, which indicates that Q can interrupt the behaviour of P if an event offered by Q is communicated.

Moreover, consider the replicated external choice of CSP $\Box\, x : A \bullet F(x)$, where x is a value from the set A, and $F(x)$ is any process expression involving x. This construction behaves as the process $F(a_1) \ \Box \ \ldots \ \Box \ F(a_k)$, for $A = \{a_1, \ldots, a_k\}$. Additionally, consider the process $RUN(s) = \Box\, ev : s \bullet ev \rightarrow RUN(s)$ that continuously offers the events from the set s.

CSP Traces Model Our test generation approach is based on refinement verification using the traces model of CSP [17]. The traces of a process P, given by $\mathcal{T}(P)$, correspond to the set of all possible sequences of events P can communicate. For instance, the traces model for the $STOP$ process is the set $\{\langle\rangle\}$; the traces of the process $a \rightarrow P$ is $\{\langle\rangle\} \cup \{\langle a \rangle \frown t \mid t \in \mathcal{T}(P)\}$; and the traces of an external choice $P \ \Box \ Q$ is $\mathcal{T}(P) \cup \mathcal{T}(Q)$. A complete definition for all CSP operators can be found in [17].

It is possible to compare the traces semantics of two processes by refinement verification using one of the CSP refinement checking tools, such as FDR [10]. A process Q refines a process P in the traces model, say $P \sqsubseteq_{\mathcal{T}} Q$, if, and only if, $\mathcal{T}(Q) \subseteq \mathcal{T}(P)$. If the refinement does not hold, FDR yields a trace (the shortest counter-example), say ce, such that $ce \in \mathcal{T}(Q)$ but $ce \notin \mathcal{T}(P)$. For instance, $(P \ \Box \ Q) \sqsubseteq_{\mathcal{T}} P$ holds, since $\mathcal{T}(P)$ is a subset of $\mathcal{T}(P \ \Box \ Q)$. However, the relation $STOP \sqsubseteq_{\mathcal{T}} accept \rightarrow STOP$ does not, since $\langle accept \rangle \in \mathcal{T}(accept \rightarrow STOP)$ but $\langle accept \rangle \notin \mathcal{T}(STOP)$. Thus, the trace $\langle accept \rangle$ is a counter-example for the above refinement expression.

Test Generation with CSP Test Purposes Test scenarios are traces from the specification that describe particular behaviours to be tested. A test case is a CSP process that is constructed from the trace, as detailed in [15]. The mapping of the CSP events of a test case into the respective CNL elements produces a test case suite that can be executed directly by a tester. In our approach, test scenarios are yielded as counter-examples of refinement verifications using CSP test purposes [14], which are CSP process representing a partial specification of the test scenarios to be selected from a specification. We summarise how CSP test purposes are used to generate test scenarios.

Let the processes $ACCEPT$, ANY and $UNTIL$ be primitive test purposes used in the construction of CSP test purposes. The process $ACCEPT = accept \rightarrow STOP$ is used to mark test scenarios by communicating the mark event $accept$, which does not belong to the alphabet of the specification process. The primitive $ANY(evset, next) = \Box\, ev : evset \bullet ev \rightarrow next$ performs basic selection. It selects the events offered by the specification that belong to $evset$. If any of these events is communicated, it behaves as $next$. Otherwise, it deadlocks. The process $UNTIL(alpha, evset, next) = RUN(alpha \setminus evset) \ \triangle \ ANY(evset, next)$, where *alpha* is the alphabet of the specification, selects all sequences offered by the specification events until it engages in some event that belongs to *evset*.

The CSP sample test purpose $TP1 = \mathit{UNTIL}(\alpha_S, \{\mathit{event}\}, \mathit{ACCEPT})$ selects test scenarios that reach the event *event* in a specification process S, such that $\mathit{event} \in \alpha_S$. Consider the expression $PP1 = S \,|[\, \alpha_S \,]|\, TP1$ that represents the parallel product between S and $TP1$. The refinement expression $S \sqsubseteq_{\tau} PP1$ yields test scenarios as counter-examples, if there are test scenarios that match $TP1$. The right-hand side of the expression behaves as the process S while the event *event* is not communicated. After the communication of *event*, the *accept* event is communicated and the composition deadlocks. Consequently, the traces ending with *accept* belong to $PP1$ but not to $\mathcal{T}(S)$. Moreover, the verification of this refinement using FDR does not hold and yields a test scenario as the shortest counter-example trace.

In general, FDR yields only the shortest counter-example trace, thus, for obtaining lengthier counter-examples it is necessary to run other verifications in an interactive manner. Further details on interactive test generation approach, and on case construction from test scenarios can be found in [15].

3 State-Based Use Case

The notation we propose for state-based use cases is introduced via a case study that describes a feature of a 4G mobile phone. The grammar for this notation is presented in [2]. The standards for mobile networking have evolved through several generations: 2^{nd} generation (2G), 3^{rd} generation (3G), and 4^{th} generation (4G). A mobile phone has three configurations related to internet over the carrier network: (i) the network mode defines which standard generation is being used; (ii) the mobile data enables or disables the access to the internet over the carrier network; and (iii) the wireless (wifi) enables or disables internet access over a wireless router. The use cases described below introduce scenarios where these configurations conflict with each other.

First, we introduce new types (Table 1). The type OnOff contains two values: On and Off, which are useful for variables that represent toggle switches. The type NetworkModes contains all possible networking compatibility standards. The N_2G and N_3G values represent connection modes that operate exclusively in a 2G and 3G network, respectively. The N_3G_2G network mode works for both 3G and 2G networks, prioritizing the 3G connection, although still attempting to connect to a 2G network whenever the 3G connection fails. Analogously, the N_4G_3G mode works for both 4G and 3G network. Finally, the N_4G_3G_2G network mode targets a 4G connection, but should work with all previous generations in the described order. Our CNL also allows a new type to be introduced either by indexing or from a base type. Indexing associates a tag with the values defined in a range. For example, the construction Phone[1,3] creates a new type composed of the values Phone.1, Phone.2 and Phone.3. A new type can also be created by associating a tag to the elements of an existing type (in general allowing disjoint unions). For instance, the new type Car.OnOff contains the values Car.On and Car.Off. Table 1 also shows the declaration of the

New Type

Id	Description	Value
OnOff	Type of a toggle switch	On, Off
NetworkModes	Types of network compatibilities	N_2G, N_3G, N_3G_2G, N_4G_3G, N_4G_3G_2G

Constants

Id	Description	Value
Non_4G_Modes	Set of non 4G modes	{N_2G, N_3G, N_3G_2G}
All_4G_Modes	Set of 4G modes	{N_4G_3G, N_4G_3G_2G}

Variables

Id	Description	VarType	Value
selectedMode	Network mode configuration	NetworkModes	N_3G_2G
mobileData	Mobile data configuration	OnOff	On
wifi	Wifi configuration	OnOff	Off

Table 1. Type, constant and variable declarations.

constants Non_4G_Modes and All_4G_Modes. The constant Non_4G_Modes is a set of NetworkModes that does not operate under 4G, while All_4G_Modes is the set complement of Non_4G_Modes.

Variables are also declared in Table 1. The variable selectedMode is of type NetworkModes and has an initial value of N_3G_2G. This variable stores the configuration of the network mode of the mobile phone. The variables mobileData and wifi are of type OnOff. When mobileData is On, the mobile phone can access the internet over the carrier network, and, when the wifi is On, the mobile phone can access the internet over a wireless router. The initial value of the mobileData is On and the wifi is Off.

Table 2 shows, at the top, the header of the Use Case 01 (UC01). It declares the use case title (Network Mode vs Mobile Data) and states that it includes the Use Case 02 (UC02) at steps 3M and 4M. We explain how the include works when introducing the UC02 in the sequel. The main flow specification of UC01 is also shown in Table 2. A step is defined by an identifier (Step ID), the user action (Action), the system state before the action is carried out (System State), and the expected result (System Response). In Step 1M, the user disables the mobile data. Regardless of the current system state, the device turns off the mobile data. Internally, the variable mobileData is assigned to Off. The notation between two occurrences of the symbol % is the subset of the CNL that manipulates state changes, inputs and outputs. In Step 2M the user selects one of the 4G network modes. The CNL sentence contained in the Action cell %**Input x : NetworkModes from All_4G_Modes**% takes an input from the user and assigns it to the local variable x. The variable x is of type NetworkModes restricted to the subset All_4G_Modes. As the mobile data was disabled in Step 1M, the system does not allow this selection to be done. The sentence (%**Output x**%) produces the value to replace the $ sign. Therefore, the message "$ is not available while mobile data is off or wifi is on." is displayed, where the $ sign is replaced by the value of x. Step 3M enables the mobile data. No change is made on the network mode (so, it remains N_3G_2G, the default value defined in Table 1), but the variable mobileData is now On. Finally, Step 4M tries again to set up a new network mode. This time, the system successfully assigns the input x to the variable selectedMode. Table 3 shows the main flow of UC02.

UC01 - Network Mode vs Mobile Data
Include

Use Case Id	Position
UC02	3M, 4M

Main Flow

Step Id	Action	System State	System Response
1M	Disable the mobile data.		The device disables the mobile data. %**mobileData := Off**%
2M	Manually select a network mode containing the 4G option. %**Input x : NetworkModes from All_4G_Modes**%		The warning "$ is not available while mobile data is off or wifi is on." is displayed and the network mode remains unchanged. %**Output x**%
3M	Enable the mobile data.		The mobile data is enabled. The selected network remains $. %**mobileData := On, Output selectedMode**%
4M	Manually select a network mode containing the 4G option. %**Input x : NetworkModes from All_4G_Modes**%		The device selects the new network mode : $ %**selectedMode := x, Output x**%

Table 2. UC01 main flow.

This use case describes the scenario where the mobile phone makes a call, sends an SMS (Short Message Service) and an MMS (Multimedia Messaging Service). A call and the sending of an SMS do not require the internet to be turned on; but an MMS does. The main flow assumes that the System State has either the mobileData or the wifi (or both) enabled. In this case, the SMS and the MMS are sent. Also, the call is made.

UC02 - Wifi and Mobile Data
Main Flow

Step Id	Action	System State	System Response
1M	Make a call, and send an SMS and an MMS.	Either the wifi or the mobile data is turned on. %**(mobileData==On) or (wifi==On)**%	The device makes the call and sends both the SMS and the MMS.

Alternative Flow
From: START
To: END

Step Id	Action	System State	System Response
1M	Make a call, and send an SMS and an MMS.	Both the wifi and the mobile data are turned off. %**(mobileData==Off) and (wifi==Off)**%	The device makes the call and sends the SMS, but is not able to send the MMS.

Table 3. UC02 main and alternative flows.

The alternative flow is also described in Table 3. The control flow jumps from the main flow START point, i.e. before step 1M happens, and returns to the main flow END point, i.e. after 1M finishes. The alternative flow introduces the scenario where no internet access is available, i.e the System State has both the mobileData and the wifi off. As a result, the call is made and the SMS is sent, but the MMS is not delivered.

The UC02 is included as part of the UC01. Table 2 shows an include declaration at steps 3M and 4M. This means that the UC02 runs twice. Firstly, it runs before step 3M and, once completed, it returns to UC01 to run step 3M. In the second time, it runs before 4M followed by the execution of 4M. So, the UC01 executes the following steps: 1M, 2M, UC02, 3M, UC02, and 4M.

4 Automatic Generation of CSP Test Models

This section describes our approach for generating CSP test models from use cases. The process which yields a CSP model is split in three steps. First, the syntax of the CNL is checked; second, an XML representing the use cases is generated; and finally, the CSP test model that represents the use case document is yielded. We further detail these steps in what follows.

Validating Use Case Documents Our use cases are described as part of a fix tabular template where its structure is well formed by construction. So part of the syntactical structure of the use cases does not need to be checked. However, we still need to verify the well-formedness of the CNL embedded in the template. So we built a parser and a type checker to carry out syntactic and semantic analysis and point out errors to the user. If no errors are reported, the process continues and we move on to the second step.

Building an XML Representation A use case is translated into an intermediate format as an XML file because it does not dependent on any formal notation. This simplifies future developments by giving the flexibility to use alternative formalisms for test generation. Figure 2 illustrates a portion of the XML schema for defining new types. The type OnOff is described in Figure 2 as an enumeration. Note that each field in Table 1 becomes a tag in XML.

Generating the CSP Model The final step yields the CSP test model, where each use case is modelled as a CSP process. Accordingly, every use case flow is also modelled as a process. If the use cases declare variables, then an additional CSP process that works as a RAM memory is created. In what follows we present the translation of some elements of the example introduced in Section 3.

The types, the constants and the variables declared in Table 1 are translated into CSP as shown in Figure 3.
The variables are mapped into a datatype *Var* (line 01). New types are mapped into the datatype *Type* (line 02), which is the disjoint union of all types. Each type is declared by a tag that identifies the type followed by the separator "." and the respective set of values (line 02). The values for the types are mapped into CSP datatypes (lines 04 and 05). For instance, $t1.On$ and $t1.Off$ are instances

```
<newType>
  <id>OnOff</id>
  <description>Type of a toggle switch</description>
  <newTypeElements>
     <enumeration>
        <element>On</element>
        <element>Off</element>
     </enumeration>
  </newTypeElements>
</newType>
```

Fig. 2. The XML for the new type OnOff.

```
01 datatype Var = wifi | mobileData | selectedMode
02 datatype Type = t1.t_OnOff | t2.t_NetworkModes
03
04 datatype t_OnOff = On | Off
05 datatype t_NetworkModes = N_2G | N_3G | N_3G_2G | N_4G_3G
06
07 Non_4G_Modes = {(N_2G), (N_3G), (N_3G_2G)}
08 All_4G_Modes = {(N_4G_3G), (N_4G_3G_2G)}
09
10 INIT = {(wifi, t1.Off), (mobileData, t1.On), (selectedMode, t2.N_3G_2G)}
11
12 channel get, set : Var.Type
```

Fig. 3. CSP model for data definitions.

of the type $t1$. Each constant is mapped to a constant in CSP (lines 07 and 08). The initial binding for the variables (Table 1) is defined as a set of pairs (variable name, initial value) named INIT (line 10). The channels *get* and *set* (line 12) communicate variable names and values. These channels are used to read the value of a variable from the memory and to update a value of a variable to the memory, respectively. For instance, the event $set.mobileData.t1.Off$ stores Off into the variable mobileData. Figure 4 shows the translation of the UC01.

The control flow is defined by the process $F1\_FLOW$ in Figure 4 (line 3), which behaves as the process $UC1\_1M$. The process $UC1\_1M$ models the step 1M of the use case UC01 (compare this process with the step 1M shown in Table 2). The event *disableData* represents the user action and the event *dataDisabled* is the system response. The event $set!mobileData!t1!Off$ updates the variable *mobileData* to the value *Off*. The event *mem_update* is a flag that indicates that a memory update has happened (this is used by the test purposes described in Section 5). The process $UC1\_1M$ behaves as *SKIP* followed by the behaviour of process $UC1\_2M$, which models the behaviour for the step 2M. The process $UC1\_2M$ uses the channels *input* and *showNotAvailable* (lines 07 and 08) to specify the values given as input, and the output value produced

```
01 channel input, showNotAvailable, showMode, modeChanged : t_NetworkModes
02
03 F1_FLOW = UC1_1M
04
05 UC1_1M = disableData → dataDisabled → set!mobileData!t1!Off →
06         mem_update → SKIP; UC1_2M
07 UC1_2M = selectMode → input?x : {e | e ∈ 4G_Modes} →
08         showNotAvailable!x → SKIP; UC1_3M
09 UC1_3M = UC2; get!selectedMode.t2?selectedMode → enableData →
10         showMode!selectedMode → set!mobileData!t1!On →
11         mem_update → SKIP; UC1_4M
12 UC1_4M = UC2; selectMode → input?x : {e | e ∈ 4G_Modes} →
13         modeChanged!x → set!selectedMode!t2!x → mem_update → SKIP
```

Fig. 4. CSP model for use case UC01.

by the system. The notation ? in $input?x : \{e \mid e \in 4G\_Modes\}$ denotes the value to be communicated is a choice of the environment. The process $UC1\_3M$ initially behaves as the process $UC2$ that specifies the behaviour of the use case UC01 (due to inclusion). Then, it reads from the memory the current value for the variable selectedMode ($get!selectedMode!t2?selectedMode$), which is communicated by the channel *showMode* as the system response (line 10). The process $UC1\_4M$ starts at UC01 because of the use case inclusion. Then it takes as input one of the 4G modes and outputs it through the channel *modeChanged*. The memory for selectedMode is updated and the process terminates succesfully (*SKIP*). We omit the model for UC02 due to space limitations.

The top-level process for the feature described in Section 3 is specified by the process $F1$, as follows.

$$\begin{aligned} F1 = ((&F1\_FLOW;\ END1) \\ &[\![\mid \alpha_{F1\_MEMORY} \cup \{success\} \mid]\!] \\ &(F1\_MEMORY \bigtriangleup END2)) \setminus (\alpha_{F1\_MEMORY} \cup \{success\}) \end{aligned}$$

This process is the parallel composition of the control flow $F1\_FLOW$ and the respective memory process $F1\_MEMORY$ synchronising on the alphabet of the memory plus the control event *success*. The process $F1\_MEMORY$ provides two services: the current value of a variable can be retrieved via the *get* event; and a new value of a variable can be stored via the *set* event. Furthermore, for each event in the form $get.var.type.val$, there is a copy $get'.var.type.val$ event, which is used for the purpose of test selection (see Section 5). The auxiliary process $END1 = willsucc \rightarrow success \rightarrow SKIP$ is used to flag successful termination via the *willsucc* event whenever a use case in $F1\_FLOW$ terminates. This process allows the successful termination of the process $F1\_FLOW$ by synchronising with the event *success* that is offered by the auxiliary process $END2 = success \rightarrow SKIP$, which interrupts the memory process whenever a use case in $F1\_FLOW$ terminates. Finally, the control events (α_{F1_MEMORY} and *success*) are hidden.

5 State-Based Test Purposes

We present the notation for authoring state-based test purposes. This language enables the partial specification of test scenarios based on the use case steps and states of the use case variables. New constructs have been added into the language to specify the selection of steps. State conditions are specified using boolean expressions. Each construct of the language is automatically translated to a CSP process that represents a test purpose, which is used for test generation in the way presented in Section 2. Thus, CSP notation is again hidden from the test designer. Table 4 presents the constructs for the test purposes notation (column Test Purpose) and the CSP processes that specify the constructs (column CSP Test Purpose).

	Test Purpose	**CSP Test Purpose**
01	*next stepId* [, *TP*]	$ANY(\{act\}, UNTIL(\alpha_S, \{res\}, T(TP)))$
02	*not stepId* [, *TP*]	$ANY(A_I \setminus \{act\}, T(TP))$
03	*stepId* [, *TP*]	$UNTIL(\alpha_S, \{res\}, T(TP))$
04	*guard* [, *TP*]	$GUARD = get'.var1?var1 \rightarrow \ldots \rightarrow get'.varN?varN \rightarrow$ $if(eval(guard))\ then\ T(TP)$ $else\ UNTIL(\alpha_S, \{mem\_update\}, GUARD)$
05	$STOP$	$UNTIL(\alpha_S, \{mem\_update\},$ $get'.var1?var1 \rightarrow \ldots \rightarrow get'.varN?varN \rightarrow$ $ACCEPT)$
06	$SUCCESS$	$willsucc \rightarrow T(STOP)$
07	$TP1\ OR\ TP2$	$T(TP1) \Box T(TP2)$
08	$TP1\ AND\ TP2$	$T(TP1) \vert[\alpha_S \cup \{accept\}]\vert T(TP2)$

Table 4. CSP model for test purposes.

Table 4 line 1 introduces the *next* construct that specifies test scenarios initiating with the step *stepId*. The notation f#uc#step is used to define *stepId*. This notation represents the full identification for a step whose id is step, and belongs to the use case uc of the feature f. For instance, F1#UC01#2M represents the step 2M of the use case UC1 that belongs to the feature F1 (in Section 3). Let the set α_S stand for the alphabet of the specification, and *act* and *res* CSP events that represent the action and the system response of the step *stepId*, respectively. The CSP model for this construct is defined in terms of the primitive test purposes ANY and $UNTIL$ introduced in Section 2. After matching *act*, the test purpose accepts any event until it finds *res*. This guarantees that the events between *act* and *res* are in the test scenario. In Table 4, $T(TP)$ stand for the CSP model for a test purpose TP. What the test purpose selects after the event *res* is (optionally) defined by a test purpose TP. If TP (preceded by a comma) is absent, it is implicitly assumed that TP is STOP (Table 4, line 5).

A test purpose in the form *not stepId*, *TP* (line 2 in Table 4) selects test scenarios that initiate with any step, except the step *stepId*. This constructor is complementary to the previous one. The CSP model for this construct selects any event in the input alphabet but *act*. The subsequent events are specified by TP. A step *stepId* not preceded by any construct (line 3), specifies test scenarios that reach the system response of *stepId*, which can be preceded by any other steps. The steps subsequent to step *stepId* are specified by TP.

Line 4 in Table 4 shows a state-based test purpose of the form $guard, TP$, where guard is a boolean expression that refers to the feature variables. The notation for describing such an expression is the same introduced in Section 3 for describing system conditions. Test scenarios in which *guard* is evaluated to true are the ones to be selected. The CSP model for $guard, TP$ initially reads the current values of the variables using get' events, then evaluates the guard. If the guard holds, the selection continues according to TP; otherwise, the test purpose looks for a memory update event (to guarantee that the state changed) and the process recurses.

The test purpose STOP (Table 4, line 5) reads all variable values and includes a mark event. Its CSP model looks for a memory update event; after that it reads the variables in order to record their values at the end of the test scenarios. Finally, it behaves as $ACCEPT$ (introduced in Section 2).

The test purpose $SUCCESS$ (line 6) selects test scenarios that terminate successfully: those for which the CSP model communicates a ✓ event. The CSP model synchronises on the event *willsucc* that flags that the use case will terminate, then it behaves as $T(STOP)$.

The last two constructs in Table 4 (lines 7 and 8) enable test purposes to be combined. Let TP1 and TP2 be test purposes. The expression TP1 OR TP2 specifies the set of test scenarios that match TP1 plus the ones that match TP2. The CSP model is the external choice of the model for TP1 with the model for TP2. Remember that the parallel product (see Section 2) is the parallel composition of a test purpose TP with the specification S. According to the laws of CSP, we have that $S \,|[\ldots]|\, T(TP1) \;\Box\; T(TP2)$ equals $S \,|[\ldots]|\, T(TP1) \;\Box\; S \,|[\ldots]|\, T(TP2)$, whose semantics is the union of $\mathcal{T}(S \,|[\ldots]|\, T(TP1))$ with $\mathcal{T}(S \,|[\ldots]|\, T(TP2))$. The expression TP1 AND TP2 specifies test scenarios that are common to TP1 and TP2. The CSP model for this expression is the parallel composition $T(TP1) \,|[\ldots]|\, T(TP2)$, whose semantics is $\mathcal{T}(T(TP1)) \cap \mathcal{T}(T(TP2))$. Consequently, this composition selects traces that are common to both test purposes.

As an example, consider the CNL test purpose TP2 = (mobileData == Off), F1#UC1#2M that selects test scenarios for which the value of the variable mobileData equals Off and that terminates in the step F1#UC1#2M, such that TP2 is the test purpose identifier (an abbreviation for mobileData == Off, . . .). Thus, the CSP model for this test purpose is characterised by the processes that follow.

$$TP2 = get'.mobileData?v \rightarrow if\,(v == t1.Off)\; then\; TP3 \\ \qquad\qquad\qquad\qquad else\; UNTIL(\alpha_{F1}, \{mem\_update\}, TP2)$$

$$TP3 = UNTIL(\alpha_{F1}, \{|\; showNotAvailable\; |\}, T(STOP))$$

$$T(STOP) = UNTIL(\alpha_{F1}, \{mem\_update\}, get'.wifi?v_1 \rightarrow \\ \qquad get'.mobileData?v_2 \rightarrow get'.selectedMode?v_3 \rightarrow ACCEPT)$$

Let $PP2 = F1 \,|[\, \alpha_{F1} \,]|\, TP2$ be the parallel product between $F1$ and $TP2$. The shortest test scenario yielded by the verification of the refinement expression $F1 \setminus \{mem\_update\} \sqsubseteq_{\mathcal{T}} PP2 \setminus \{mem\_update\}$ is presented in the sequel, namely ts. The hiding of the control event mem_update removes such an event from the yielded test scenarios.

$$ts = \langle\; get'.mobileData.t1.On, disableData, dataDisable, \\ get'.mobileData.t1.Off, selecteMode, input.N\_4G\_3G, \\ showNotAvailable.N\_4G\_3G, get'.wifi.t1.Off, \\ get'.mobileData.t1.Off, get'.selectedMode.t2.N\_3G\_2G, accept\rangle$$

As expected, the get' events in the end of ts show that mobileData equals Off after the system output in F1#UC01#2M (event $showNotAvailable.N\_4G\_3G$). The events in ts are translated to CNL counterparts with the reverse mapping from CNL to CSP used in Section 4 for generating the CSP model. This yields a CNL test case that is used for manual execution.

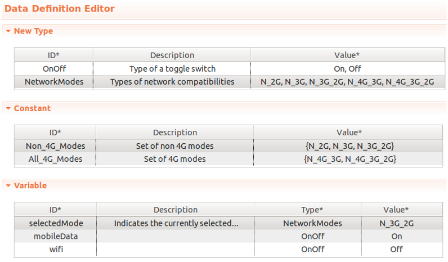
6 Tool support

This section presents an overview of the TaRGeT tool with the necessary extensions to mechanise the entire strategy described in this paper, and, particularly, it gives a short demonstration of how the authoring of state-based use cases and test purposes is supported by the tool.

TaRGeT is developed using Java [16] and Eclipse RCP [13] with the idea to aggregate different capabilities in separate implementation units called plugins. A new plugin was added to support the new GUI interfaces for authoring state-based use cases and CNL test purpose . It also comprises a compiler that validates the CNL notation and translates from CNL to CSP, and a test generator component which calls FDR to generate test cases.

Data Definition Editor Figure 5 shows the interface of the data definition editor, created to assist the test designer in the task of authoring data in our use case template. Several verifications are performed during the edition as syntactic and semantic checks to report errors.

Test Purpose Generation TaRGeT has also been extended to consider state-based test purposes. Figure 6 illustrates the interface which allows the definition of test purposes. Each test purpose is given a unique identifier. This allows the user to create new test purposes composed of previously defined ones.



Data Definition Editor

New Type

ID*	Description	Value*
OnOff	Type of a toggle switch	On, Off
NetworkModes	Types of network compatibilities	N_2G, N_3G, N_3G_2G, N_4G_3G, N_4G_3G_2G

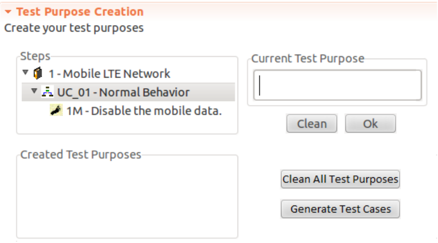
Constant

ID*	Description	Value*
Non_4G_Modes	Set of non 4G modes	{N_2G, N_3G, N_3G_2G}
All_4G_Modes	Set of 4G modes	{N_4G_3G, N_4G_3G_2G}

Variable

ID*	Description	Type*	Value*
selectedMode	Indicates the currently selected...	NetworkModes	N_3G_2G
mobileData		OnOff	On
wifi		OnOff	Off

Fig. 5. The data definition editor.



Test Purpose Creation

Create your test purposes

Steps

- 1 - Mobile LTE Network
  - UC_01 - Normal Behavior
    - 1M - Disable the mobile data.

Fig. 6. Test purpose creation

After the user finishes editing test purposes, the extended version of TaRGeT converts use cases to XML, which is translated to CSP as detailed in Section 4. Test purposes, after syntactic and semantic checks, are converted directly to CSP processes that are used for generating test scenarios as explained in Section 5. Finally, test scenarios are translated to test cases for test execution.

7 Conclusion

Use cases are widely used in industry in consolidated approaches to requirements specification. A use case specification typically describes, in the form of an event flow, in natural language, interactions between the user and the system. Unfortunately, natural language is hard to analyse and to process. In this work, we have proposed an approach to specify use cases in a Controlled Natural Language. Our use cases are able to specify both control flow and data aspects. The control flow is captured via the tabular structure of our use cases in addition to commands that take the flow to alternative, inclusion and extension of use cases. The data flow is embedded in the use case specification via the CNL for manipulating inputs, outputs and state variables.

Both control and data constructs are translated into a CSP model. By adopting CSP we can take advantage of tools like FDR to mechanise test generation, which is carried out as a process refinement computation: the counter-example produced by FDR is used to build a test case. In order to select and control which test cases should be generated, we have also shown how the CNL can be used to specify test purposes, which are also translated into CSP; once it runs in parallel with the CSP translation of the use cases, it automatically selects which test cases are produced. A tool fully integrated with FDR has been developed for use case editing and test case generation using test purposes.

Traditionally, automatic test case generation is carried out from formal languages that are typically not mastered by test designers; therefore this demands extra effort in the testing process: the test designer needs to translate the requirements to the formal language and the test cases produced back to another format (either natural language or a programming language). With a Controlled Natural Language embedded on a widely used requirements artifact, such as use cases, we conceal the formal aspects of test case generation.

Several works have proposed approaches to generate formal specifications from requirements to solve the problem of ambiguous, imprecise and unclear specifications [4, 8, 20–22]. However, differently from our approach, the formal notations used are not a process algebra, their formal models are not used with the purpose of test derivation, and the input, output and state are not analysed and processed as we do. For instance, Schwitter and Fuchs [20] propose a Controlled Natural Language that resembles a formal specification in logic and that can be translated to Prolog [24]. The notions of input, output and state that may appear in the CNL are translated as Prolog facts and not processed as part of a state based requirements. Similarly, Sinha *et al.* [21] analyse use cases for the purpose of edit-time, instantaneous inspection of their style and content. Inputs, outputs, and states are not processed as part of a system, but as part of a text to be checked for completeness, structure, flow, dependencies, etc. Some other works introduce approaches for test case generation from requirements described in a variety of notations [3, 5, 7, 23, 6]. Requirements are translated to a model that is used as the input for test generation. For instance, Carvalho *et al.* [7] introduce an approach for the generation of test vectors from a Controlled Natural Language that captures temporal requirements for the domain of controlling systems. Transition relations are used as the formal specification language. Somé and Cheng [23] take as input use cases written in natural language with a well defined syntax. Similarly to our work, their approach allows the use of step flows, condition and operations within the test cases. However, there is no notion of inputs and outputs and it is only possible to use data types provided by default. New types cannot be created by the user as in our approach. Another related approach is [6] that generates test cases for control system based on a natural language that supports timed reactive system. None of these approaches allow the specification of the test purposes in a Controlled Natural Language.

As future work, we intend to investigate how to produce executable test cases, particularly for the UIAutomator framework [1]. We also plan to explore the use of compression techniques of FDR and analyse efficiency gains in refinement verification. Concerning scalability, as a future work we plan to apply our strategy to more complex systems.

References

1. Android Developers: UIAutomator. (2015), https://developer.android.com/tools/testing-support-library/index.html
2. Bezerra, R.: Extração Automática de Modelos CSP a partir de Casos de Uso. Master's thesis, Center of Informatics of Federal University of Pernambuco (2011)
3. Boddu, R et al.: RETNA: From requirements to testing in a natural way. In: Proceedings of the RE 2004. pp. 262–271. IEEE, Washington, DC, USA (2004)

4. Brottier, E., Baudry, B., Traon, Y.L., Touzet, D., Nicolas, B.: Producing a global requirement model from multiple requirement specifications. In: Proceedings of the 11th EDOC. pp. 390–. EDOC '07, IEEE, Washington, DC, USA (2007)
5. C Nebut et al.: Automatic test generation: a use case driven approach. IEEE Transactions on Software Engineering 32(3), 140–155 (2006)
6. Carvalho, G. et al.: Test case generation from natural language requirements based on scr specifications. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. pp. 1217–1222. SAC '13, ACM, New York, NY, USA (2013)
7. Carvalho, G. et al.: Model-based testing from controlled natural language requirements. In: Artho, C., Ölveczky, P.C. (eds.) Formal Techniques for Safety-Critical Systems, CCIS, vol. 419, pp. 19–35. Springer International (2014)
8. Drazan, J., Mencl, V.: Improved processing of textual use cases: Deriving behavior specifications. In: SOFSEM 2007: Theory and Practice of Computer Science, LNCS, vol. 4362, pp. 856–868. Springer Berlin Heidelberg (2007)
9. Ferreira, F., Neves, L., Silva, M., Borba, P.: TaRGeT: a Model Based Product Line Testing Tool. In: Proceedings of CBSoft 2010 — Tools Panel (2010)
10. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 17 A Modern Refinement Checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 187–201 (2014)
11. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
12. Jard, C., Jéron, T.: TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. International Journal of Software Technology Transfer 7(4), 297–315 (2005)
13. McAffer, Jeff et al.: Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications. Addison-Wesley Professional (2005)
14. Nogueira, S., Sampaio, A., Mota, A.: Guided Test Generation from CSP Models. In: Proceedings of the 5th ICTAC. pp. 258–273. Springer, Berlin, Heidelberg (2008)
15. Nogueira, S., Sampaio, A., Mota, A.: Test generation from state based use case models. Formal Aspects of Computing 26(3), 441–490 (2014)
16. Oracle: Java JSE. http://www.oracle.com/ (July 2015)
17. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall PTR (1998)
18. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science, Springer (2010)
19. Santiago Júnior, V.A.D., Vijaykumar, N.L.: Generating model-based test cases from natural language requirements for space application software. Software Quality Journal 20(1), 77–143 (Mar 2012)
20. Schwitter, R., Fuchs, N.E.: Attempto - from specifications in controlled natural language towards executable specifications. CoRR cmp-lg/9603004 (1996)
21. Sinha, A., Jr., S.M.S., Paradkar, A.: Text2test: Automated inspection of natural language use cases. In: Proceedings of the ICST 2010. pp. 155–164. ICST '10, IEEE Computer Society, Washington, DC, USA (2010)
22. Somé, S.S.: Supporting use case based requirements engineering. Information and Software Technology 48(1), 43 – 58 (2006)
23. Somé, S.S., Cheng, X.: An approach for supporting system-level test scenarios generation from textual use cases. In: Proceedings of SAC 2008. pp. 724–729. ACM, New York, NY, USA (2008)
24. Sterling, L., Shapiro, E.: The Art of Prolog. MIT Press, Cambridge, Massachusetts (1986)
25. Veanes, M., Campbell, C., Schulte, W., Tillmann, N.: Online testing with model programs. In: Proceedings of the 10th European software engineering conference. pp. 273–282. ESEC/FSE-13, ACM, New York, NY, USA (2005)

A Mechanized Textbook Proof of a Type Unification Algorithm

Rodrigo Ribeiro[1] and Carlos Camarão[2]

[1] Universidade Federal de Ouro Preto, João Monlevade, Minas Gerais, Brazil
`rodrigo@decsi.ufop.br`,
[2] Universidade Federal de Minas Gerais, Belo Horizonte, Minas Gerais, Brazil
`camarao@dcc.ufmg.br`

Abstract. Unification is the core of type inference algorithms for modern functional programming languages, like Haskell. As a first step towards a formalization of a type inference algorithm for such programming languages, we present a formalization in Coq of a type unification algorithm that follows classic algorithms presented in programming language textbooks.

1 Introduction

Modern functional programming languages like Haskell [1] and ML [2] provide type inference to free the programmer from having to write (almost all) type annotations in programs. Compilers for these languages can discover missing type information through a process called type inference [3].

Type inference algorithms are usually divided into two components: constraint generation and constraint solving [4]. For languages that use ML-style (or parametric) polymorphism, constraint solving reduces to first order unification.

A sound and complete algorithm for first order unification is due to Robinson [5]. The soundness and completeness proofs have a constructive nature, and can thus be formalized in proof assistant systems based on type theory, like Coq [6] and Agda [7]. Formalizations of unification have been reported before in the literature [8–11] using different proof assistants, but none of them follows the style of textbook proofs (cf. e.g. [12, 13]).

As a first step towards a full formalization of a type inference algorithm for Haskell, in this article, we describe an axiom-free formalization of type unification in the Coq proof assistant, that follows classic algorithms on type systems for programming languages [12, 13]. The formalization is "axiom-free" because it does not depend on axioms like function extensionality, proof irrelevance or the law of the excluded middle, i.e. our results are integrally proven in Coq.

More specifically, our contributions are:

1. A mechanization of a termination proof as it can be found in e.g. [12, 13]. In these books, the proof is described as "easy to check". In our formalization, it was necessary to decompose the proof in several lemmas in order to convince Coq's termination checker.

2. A correct by construction formalization of unification. In our formalization the unification function has a dependent type that specifies that unification produces the most general unifier of a given set of equality constraints, or a proof that explains why this set of equalities does not have a unifier (i.e. our unification definition is a view [14] on lists of equality constraints).

We chose Coq to develop this formalization because it is an industrial strength proof assistant that has been used in several large scale projects such as a Certified C compiler [15], a Java Card platform [16] and on verification of mathematical theorems (cf. e.g. [17, 18]).

The rest of this paper is organized as follows. Section 2 presents a brief introduction to the Coq proof assistant. Section 3 presents some definitions used in the formalization. Section 4 presents the unification algorithm. Termination, soundness and completeness proofs are described in Sections 4.1 and 4.2, respectively. Section 5 presents details about proof automation techniques used in our formalization. Section 6 presents related work and Section 7 concludes.

While all the code on which this paper is based has been developed in Coq, we adopt a "lighter" syntax in the presentation of its code fragments. In the introductory Section 2, however, we present small Coq source code pieces. We chose this presentation style in order to improve readability, because functions that use dependently typed pattern matching require a high number of type annotations, that would deviate from our objective of providing a formalization that is easy to understand. For theorems and lemmas, we sketch the proof strategy but omit tactic scripts. The developed formalization was verified using Coq version 8.4 and it is available online [19].

2 A Taste of Coq Proof Assistant

Coq is a proof assistant based on the calculus of inductive constructions (CIC) [6], a higher order typed λ-calculus extended with inductive definitions. Theorem proving in Coq follows the ideas of the so-called "BHK-correspondence"[3], where types represent logical formulas, λ-terms represent proofs [20] and the task of checking if a piece of text is a proof of a given formula corresponds to checking if the term that represents the proof has the type corresponding to the given formula.

However, writing a proof term whose type is that of a logical formula can be a hard task, even for very simple propositions. In order to make the writing of complex proofs easier, Coq provides *tactics*, which are commands that can be used to construct proof terms in a more user friendly way.

As a tiny example, consider the task of proving the following simple formula of propositional logic:

$$(A \to B) \to (B \to C) \to A \to C$$

[3] Abbreviation of Brouwer, Heyting, Kolmogorov, de Bruijn and Martin-Löf Correspondence. This is also known as the Curry-Howard "isomorphism".

In Coq, such theorem can be expressed as:

```
Section EXAMPLE.
   Variables A B C : Prop.
   Theorem example : (A -> B) -> (B -> C) -> A -> C.
   Proof.
       intros H H' HA. apply H'. apply H. assumption.
   Qed.
End EXAMPLE.
```

In the previous source code piece, we have defined a Coq section named `EXAMPLE`[4] which declares variables `A`, `B` and `C` as being propositions (i.e. with type `Prop`). Tactic `intros` introduces variables `H`, `H'` and `HA` into the (typing) context, respectively with types `A -> B`, `B -> C` and `A` and leaves goal `C` to be proved. Tactic `apply`, used with a term `t`, generates goal `P` when there exists `t: P -> Q` in the typing context and the current goal is `Q`. Thus, `apply H'` changes the goal from `C` to `B` and `apply H` changes the goal to `A`. Tactic `assumption` traverses the typing context to find a hypothesis that matches with the goal.

We define next a proof of the previous propositional logical formula that, in contrast to the previous proof, that was built using tactics (`intros`, `apply` and `assumption`), is coded directly as a function:

```
   Definition example' : (A -> B) -> (B -> C) -> A -> C :=
       fun (H : A -> B) (H' : B -> C) (HA : A) => H' (H HA).
```

However, even for very simple theorems, coding a definition directly as a Coq term can be a hard task. Because of this, the use of tactics has become the standard way of proving theorems in Coq. Furthermore, the Coq proof assistant provides not only a great number of tactics but also a domain specific language for scripted proof automation, called $\mathcal{L}$tac. In this work, the developed proofs follow the style advocated by Chlipala [21], where most proofs are built using $\mathcal{L}$tac scripts, to automate proof steps and make them more robust. Details about $\mathcal{L}$tac can be found in [21, 6].

3 Definitions

3.1 Types

We consider a language of simple types formed by type variables, type constants (also called type constructors) and functional types given by the following grammar:

$$\tau ::= \alpha \mid c \mid \tau \to \tau$$

where α stands for a type variable and c a type constructor. All meta-variables (τ, α and c) can appear primed or subscripted and as usual we consider that $\to$ associates to the right.

Identifiers for variables and constructors are represented as natural numbers, following standard practice in formalized meta-theory [22, 23]. We are aware that

[4] In Coq, we can use sections to delimit the scope of local variables.

choosing this representation of types is not adequate to represent Haskell's types, since it does not allow the occurrence of n-ary type constructors. Using n-ary type constructors will only clutter definitions due to the need of using kinds[5]. Since the presence of kind information is orthogonal to unification, we prefer to omit it in order to clarify definitions and proofs.

The list of type variables of type τ is denoted by $\mathrm{FV}(\tau)$.

The *size* of a given type τ, given by the number of arrows, type variables and constructors in τ, is denoted by $size(\tau)$. Formally:

$$\begin{array}{ll} size(\tau_1 \to \tau_2) & = 1 + size(\tau_1) + size(\tau_2) \\ size(\tau) & = 1 \text{ otherwise } (\tau = \alpha \text{ or } \tau = c\text{, for some } \alpha, c) \end{array}$$

We let $\tau_1 \overset{e}{=} \tau_2$ denote the equality constraint between two types τ_1 and τ_2.

Lists of equality constraints are represented by meta-variable $\mathbb{C}$. We use the left-associative operator :: for constructing lists: $a :: x$ denotes the list formed by head a and tail x.

The definition of free type variables for constraints and their lists are defined in a standard way and the size of constraints and constraint lists are defined as the sum of their constituent types. The following simple lemmas will be later used to establish termination of the unification algorithm, defined in Section 4.

Lemma 1 *For all types $\tau_1, \tau_1', \tau_2, \tau_2'$ and all lists of constraints $\mathbb{C}$ we have that:*

$$size((\tau_1 \overset{e}{=} \tau_1') :: (\tau_2 \overset{e}{=} \tau_2') :: \mathbb{C}) < size((\tau_1 \to \tau_2 \overset{e}{=} \tau_1' \to \tau_2') :: \mathbb{C})$$

Proof. Induction over $\mathbb{C}$ using the definition of *size*.

Lemma 2 *For all types τ, τ' and all lists of constraints $\mathbb{C}$ we have that*

$$size(\mathbb{C}) < size((\tau \overset{e}{=} \tau') :: \mathbb{C})$$

Proof. Induction over τ and case analysis over τ', using the definition of *size*.

3.2 Substitutions

Substitutions are functions mapping type variables to types. For convenience, a substitution is considered as a finite mapping $[\alpha_1 \mapsto \tau_1, ..., \alpha_n \mapsto \tau_n]$, for $i = 1, \ldots, n$, which is also abbreviated as $[\overline{\alpha} \mapsto \overline{\tau}]$ ($\overline{\alpha}$ and $\overline{\tau}$ denoting sequences built from sets $\{\alpha_1, ..., \alpha_n\}$ and $\{\tau_1, ..., \tau_n\}$, respectively). Meta-variable S is used to denote substitutions.

In our formalization, a mapping $[\alpha \mapsto \tau]$ is represented as a pair of a variable and a type. Substitutions are represented as lists of mappings, taking advantage

[5] Kinds classify type expressions in the same way as types classify terms. More details about the use of kinds and high-order operators can be found in [13].

of the fact that a variable never appears twice in a substitution. The domain of a substitution, denoted by $dom(S)$, is defined as:

$$dom(S) = \{\alpha \mid S(\alpha) = \tau, \alpha \neq \tau\}$$

Following [10], we define *substitution application* in a variable-by-variable way; first, let the application of a mapping $[\alpha \mapsto \tau']$ to τ be defined by recursion over the structure of τ:

$$\begin{array}{ll}
[\alpha \mapsto \tau'] \, (\tau_1 \to \tau_2) & = ([\alpha \mapsto \tau'] \, \tau_1) \to ([\alpha \mapsto \tau'] \, \tau_2) \\
[\alpha \mapsto \tau'] \, \alpha & = \tau' \\
[\alpha \mapsto \tau'] \, \tau & = \tau \;\; \text{otherwise } (\tau = \alpha' \text{ for some } \alpha' \neq \alpha, \text{ or} \\
 & \qquad\qquad\qquad\quad \tau = c \text{ for some } c)
\end{array}$$

Next, substitution application follows by recursion on the number of mappings of the substitution, using the above defined application of a single mapping:

$$S(\tau) = \begin{cases} \tau & \text{if } S = [\,] \\ S'([\alpha \mapsto \tau'] \, \tau) & \text{if } S = [\alpha \mapsto \tau'] :: S' \end{cases}$$

Application of a substitution to an equality constraint is defined in a straightfoward way:

$$S \; (\tau \overset{e}{=} \tau') = S(\tau) \overset{e}{=} S(\tau')$$

In order to maintain our development on a fully constructive ground, we use the following lemma, to cater for proofs of equality of substitutions. This lemma is used to prove that the result of the unification algorithm yields the most general unifier of a given set of types.

Lemma 3 *For all substitutions S and S', if $S(\alpha) = S'(\alpha)$ for all variables α, then $S(\tau) = S'(\tau)$ for all types τ.*

Proof. Induction over τ, using the definition of substitution application.

Substitutions and types are subject to well-formedness conditions, described in the next section.

3.3 Well-Formedness Conditions

Now, we consider notions of well-formedness with regard to types, substitutions and constraints. These notions are crucial to give simple proofs for termination, soundness and completeness of the unification algorithm.

Well-formed conditions are expressed in terms of a type variable context, $\mathcal{V}$, that contains, in each step of the execution of the unification algorithm, the *complement* of the set of type variables that are in the domain of the unifier. This context is used to formalize some notions that are assumed as immediate facts in textbooks, like: “at each recursive call of the unification algorithm, the number of distinct type variables occurring in constraints decreases” or “after applying a substitution S to a given type τ, we have that $FV(S(\tau)) \cap dom(S) = \emptyset$”.

We consider that:

– A type τ is well-formed in $\mathcal{V}$, written as $wf(\mathcal{V}, \tau)$, if all type variables that occur in τ are in $\mathcal{V}$.
– A constraint $\tau_1 \overset{e}{=} \tau_2$ is well-formed, written as $wf(\mathcal{V}, \tau_1 \overset{e}{=} \tau_2)$, if both τ_1 and τ_2 are well-formed in $\mathcal{V}$.
– A list of constraints $\mathbb{C}$ is well-formed in $\mathcal{V}$, written as $wf(\mathcal{V}, \mathbb{C})$, if all of its equality constraints are well-formed in $\mathcal{V}$.
– A substitution $S = \{[\alpha \mapsto \tau]\} :: S'$ is well-formed in $\mathcal{V}$, written as $wf(\mathcal{V}, S)$, if the following conditions apply:
  - $\alpha \in \mathcal{V}$
  - $wf(\mathcal{V} - \{\alpha\}, \tau)$
  - $wf(\mathcal{V} - \{\alpha\}, S')$

The requirement that type τ is well-formed in $\mathcal{V} - \{\alpha\}$ is necessary in order for $[\alpha \mapsto \tau]$ to be a well-formed substitution. This avoids cyclic equalities that would introduce infinite type expressions.

The well-formedness conditions are defined as recursive Coq functions that compute dependent types from a given variable context and a type, constraint or substitution.

A first application of these well-formedness conditions is to enable a simple definition of composition of substitutions. Let S_1 and S_2 be substitutions such that $wf(\mathcal{V}, S_1)$ and $wf(\mathcal{V} - dom(S_1), S_2)$. The composition $S_2 \circ S_1$ can be defined simply as the append operation of these substitutions:

$$S_2 \circ S_1 = S_1 \mathbin{{+}{+}} S_2$$

The idea of indexing substitutions by type variables that can appear in its domain and its use to give a simple definition of composition was proposed in [10].

We say that a substitution S is more general than S', written as $S \leq S'$, if there exists a substitution S_1 such that $S' = S_1 \circ S$.

The definition of composition of substitutions satisfies the following theorem:

Theorem 1 (Substitution Composition and Application) *For all types τ and all substitutions S_1, S_2 such that $wf(\mathcal{V}, S_1)$ and $wf(\mathcal{V} - dom(S_1), S_2)$ we have that $(S_2 \circ S_1)(\tau) = S_2(S_1(\tau))$.*

Proof. By induction over the structure of S_2.

3.4 Occurs Check

Type unification algorithms use a well-known occurs check in order to avoid the generation of cyclic mappings in a substitution, like $[\alpha \mapsto \alpha \to \alpha]$. In the context of finite type expressions, cyclic mappings do not make sense. In order to define the occurs check, we first define a dependent type, $occurs(\alpha, \tau)$, that is inhabited[6] only if $\alpha \in \mathrm{FV}(\tau)$:

[6] According to the BHK-interpretation, a type is inhabited only if it represents a logic proposition that is provable.

$$\begin{array}{ll} \mathit{occurs}(\alpha, \tau_1 \to \tau_2) & = \mathit{occurs}(\alpha, \tau_1) \vee \mathit{occurs}(\alpha, \tau_2) \\ \mathit{occurs}(\alpha, \alpha) & = \texttt{True} \\ \mathit{occurs}(\alpha, \tau) & = \texttt{False} \text{ otherwise} \\ & \qquad \text{i.e. if } \tau = \alpha' \text{ for some } \alpha' \neq \alpha \text{ or } \tau = c \text{ for some } c \end{array}$$

Coq types `True` and `False` are the unit and empty type[7], respectively. Note that $\mathit{occurs}(\alpha, \tau)$ is provable if and only if $\alpha \in \mathrm{FV}(\tau)$.

Using type *occurs*, decidability of the occurs check can be established, by using the following theorem:

Lemma 4 (Decidability of occurs check) *For all variables α and all types τ, we have that either $\mathit{occurs}(\alpha, \tau)$ or $\neg\, \mathit{occurs}(\alpha, \tau)$ holds.*

Proof. Induction over the structure of τ.

If a variable α does not occur in a well-formed type, this type is well-formed in a variable context where α does not occur. This simple fact is an important step used to prove termination of unification. The next lemmas formalize this notion.

Lemma 5 *For all variables α_1, α_2 and all variable contexts $\mathcal{V}$, if $\alpha_1 \in \mathcal{V}$ and $\alpha_2 \neq \alpha_1$ then $\alpha_1 \in (\mathcal{V} - \{\alpha_2\})$.*

Proof. Induction over $\mathcal{V}$.

Lemma 6 *Let τ be a well-formed type in a variable context $\mathcal{V}$ and let α be a variable such that $\neg \mathit{occurs}(\alpha, \tau)$. Then τ is well-formed in $\mathcal{V} - \{\alpha\}$.*

Proof. Induction on the structure of τ, using Lemma 5 in the variable case.

4 The Unification Algorithm

We use the following standard presentation of the first-order unification algorithm, where $\tau \equiv \tau'$ denotes a decidable equality test between τ and τ':

Our formalization differs from the presented algorithm (Figure 1) in two aspects:

- Since this presentation of the unification algorithm is general recursive, i.e., the recursive calls aren't necessarily made on structurally smaller arguments, we need to define it using recursion on proofs that *unify*'s arguments form a well-founded relation [6].

[7] In type theory terminology, the unit type is a type that has a unique inhabitant and the empty type is a type that does not have inhabitants. Under BHK-interpretation, they correspond to a true and false propositions, respectively [20].

(1) $\mathit{unify}([\,]) = [\,]$
(2) $\mathit{unify}((\alpha \overset{e}{=} \alpha) :: \mathbb{C}) = \mathit{unify}(\mathbb{C})$
(3) $\mathit{unify}((\alpha \overset{e}{=} \tau) :: \mathbb{C}) = \text{if } \mathit{occurs}(\alpha, \tau) \text{ then fail else } \mathit{unify}([\alpha \mapsto \tau]\mathbb{C}) \circ [\alpha \mapsto \tau]$
(4) $\mathit{unify}((\tau \overset{e}{=} \alpha) :: \mathbb{C}) = \text{if } \mathit{occurs}(\alpha, \tau) \text{ then fail else } \mathit{unify}([\alpha \mapsto \tau]\mathbb{C}) \circ [\alpha \mapsto \tau]$
(5) $\mathit{unify}((\tau_1 \rightarrow \tau_2 \overset{e}{=} \tau \rightarrow \tau') :: \mathbb{C}) = \mathit{unify}((\tau_1 \overset{e}{=} \tau) :: (\tau_2 \overset{e}{=} \tau') :: \mathbb{C})$
(6) $\mathit{unify}((\tau \overset{e}{=} \tau') :: \mathbb{C}) = \text{if } \tau \equiv \tau' \text{ then } \mathit{unify}(\mathbb{C}) \text{ else fail}$

Fig. 1. Unification algorithm.

– Instead of returning just a substitution that represents the argument constraint unifier, we return a proof that such substitution is indeed its most general unifier or a proof explaining that such unifier does not exist, when *unify* fails.

These two aspect are discussed in Sections 4.1 and 4.2, respectively.

It is worth mentioning that there are some Coq extensions that make the definitions of general recursive functions and functions defined by pattern matching on dependent types easier, namely commands `Function` and `Program`, respectively. However, according to [24], these are experimental extensions. Thus, we prefer to use well established approaches to overcome these problems: 1) use of a recursion principle derived from the definition of a well-founded relation [6] and 2) annotate every pattern matching construct in order to make explicit the relation between function argument and return types.

4.1 Termination Proof

The unification algorithm always terminates for any list of equalities, either by returning their most general unifier or by establishing that there is no unifier. The termination argument uses a notion of *degree* of a list of constraints $\mathbb{C}$, written as $\mathit{degree}(\mathbb{C})$, defined as a pair (m, n), where m is the number of distinct type variables in $\mathbb{C}$ and n is the total size of the types in $\mathbb{C}$. We let $(n, m) \prec (n', m')$ denote the usual lexicographic ordering of degrees.

Textbooks usually consider it "easy to check" that each clause of the unification algorithm either terminates (with success or failure) or else make a recursive call with a list of constraints that has a lexicographically smaller degree. Since the implemented unification function is defined by recursion over proofs of lexicographic ordering of degrees, we must ensure that all recursive calls are made on smaller lists of constraints. In lines 3 and 4 of Figure 1, the recursive calls are made on a list of constraints of smaller degree, because the list of constraints $[\alpha \mapsto \tau]\mathbb{C}$ will decrease by one the number of type variables occurring in it. This is formalized in the following lemma:

Lemma 7 (Substitution application decreases degree) *For all variables $\alpha \in \mathcal{V}$, all well-formed types τ and well-formed lists of constraints $\mathbb{C}$, it holds* *that*

$$degree([\alpha \mapsto \tau]\,\mathbb{C}) \prec degree((\alpha \overset{e}{=} \tau) :: \mathbb{C})$$

Proof. Induction over $\mathbb{C}$.

On line 5 of Figure 1, we have that the recursive call is made on a constraint that has more equalities than the original but has a smaller degree, as shown by the following lemma.

Lemma 8 (Fewer Arrows implies lower degree) *For all well-formed types $\tau_1, \tau_2, \tau_1', \tau_2'$ and all well-formed lists of constraints $\mathbb{C}$, it holds that*

$$degree((\tau_1 \overset{e}{=} \tau_1', \tau_2 \overset{e}{=} \tau_2') :: \mathbb{C}) \prec degree((\tau_1 \to \tau_2 \overset{e}{=} \tau_1' \to \tau_2') :: \mathbb{C})$$

Proof. Immediate from Lemma 1.

Finally, the recursive calls in lines 2 and 6 also decrease the degree of the input list of constraints, according to the following:

Lemma 9 (Less constraints implies lower degree) *For all well-formed types τ, τ' and all well-formed list of constraints $\mathbb{C}$, it holds that*

$$degree(\mathbb{C}) \prec degree(\{\tau \overset{e}{=} \tau'\} :: \mathbb{C})$$

Proof. Immediate from Lemma 2.

4.2 Soundness and Completeness Proof

Given an arbitrary list of constraints, the unification algorithm either fails or returns its most general unifier. We have the following properties:

- Soundness: the substitution produced is a unifier of the constraints.
- Completeness: the returned substitution is the least unifier, according to the substitution ordering defined in Section 3.2.

A substitution S is called a unifier of a list of constraints $\mathbb{C}$ according to whether $unifier(\mathbb{C}, S)$ is provable, where $unifier(\mathbb{C}, S)$ is defined by induction on $\mathbb{C}$ as follows:

$$\begin{aligned} &unifier([], S) &&= \texttt{True} \\ &unifier((\tau \overset{e}{=} \tau') :: \mathbb{C}', S) &&= S(\tau) = S(\tau') \land unifier(\mathbb{C}', S) \end{aligned}$$

A substitution S is a most general unifier of a list of constraints $\mathbb{C}$ if, for any other unifier S' of $\mathbb{C}$, there exists S_1 such that $S' = S_1 \circ S$; formally:

$$least(S, \mathbb{C}) = \forall S'.\, unifier(\mathbb{C}, S') \to \exists S_1.\, \forall \alpha.\, (S_1 \circ S)(\alpha) = S'(\alpha)$$

The type of the unification function is a dependent type that ensures the following property of the returned substitution S:

$$\big(\mathit{unifier}(\mathbb{C}, S) \wedge \mathit{least}(S, \mathbb{C})\big) \vee \text{UnifyFailure}(\mathbb{C})$$

where UnifyFailure($\mathbb{C}$) is a type that encodes the reason why unification of $\mathbb{C}$ fails. There are two possible causes of failure: 1) an occurs check error, 2) an error caused by trying to unify distinct type constructors.

In the formalization source code, the definition of the *unify* function contains "holes"[8] to mark positions where proof terms are expected. Instead of writing such proof terms, we left them unspecified and use tactics to fill them with appropriate proofs. In the companion source code, the unification function is full of such holes and they mark the position of proof obligations for soundness, completeness and termination for each equation of the definition of *unify*.

In order to prove soundness obligations we define several small lemmas that are direct consequences of the definition of the application of substitutions, which are omitted for brevity. Other lemmas necessary to ensure soundness are sketched below. They specify properties of unification and application of substitutions.

Lemma 10 *For all type variables α, types τ, τ' and substitutions S, if $S(\alpha) = S(\tau')$ then $S(\tau) = S([\alpha \mapsto \tau']\,\tau)$.*

Proof. Induction over the structure of τ.

Lemma 11 *For all type variables α, types τ, variable contexts $\mathcal{V}$ and constraint sets $\mathbb{C}$, if $S(\alpha) = S(\tau)$ and $\mathit{unifier}(\mathbb{C}, S)$ then $\mathit{unifier}([\alpha \mapsto \tau]\,\mathbb{C}, S)$.*

Proof. Induction over $\mathbb{C}$ using Lemma 10.

Completeness proof obligations are filled by scripted automatic proof tactics using Lemma 3.

5 Automating Proofs

Most parts of most proofs used to prove properties of programming languages and of algorithms are exercises that consist of a lot of somewhat tedious steps, with just a few cases representing the core insights. It is not unusual for mechanized proofs to take significant amounts of code on uninteresting cases and quite significant effort on writing that code. In order to deal with this problem in our development, we use $\mathcal{L}$tac, Coq's domain specific language for writing custom tactics, and Coq built-in automatic tactic `auto`, which implements a Prolog-like resolution proof construction procedure using hint databases within a depth limit.

The main $\mathcal{L}$tac custom tactic used in our development is a proof state simplifier that performs several manipulations on the hypotheses and on the conclusion.

[8] A hole in a function definition is a subterm that is left unspecified. In Coq, holes are represented by underscores and such unspecified parts of a definition are usually filled by tactic generated terms.

It is defined by means of two tactics, called `mysimp` and `s`. Tactic `mysimp` tries to reduce the goal and repeatedly applies tactic `s` to the proof state until all goals are solved or a failure occurs.

Tactic `s`, shown in Figure 2, performs pattern matching on a proof state using $\mathcal{L}$tac `match goal` construct. Patterns have the form:

$$[\mathtt{h}_1\ :\ \mathtt{t}_1,\mathtt{h}_2\ :\ \mathtt{t}_2\ \ldots\ \mathtt{|-}\ \mathtt{C}\ ]\ \mathtt{=>}\ \mathtt{tac}$$

where each of $\mathtt{t}_i$ and `C` are expressions, which represents hypotheses and conclusion, respectively, and `tac` is the tactic that is executed when a successful match occurs. Variables with question marks can occur in $\mathcal{L}$tac patterns, and can appear in `tac` without the question mark. Names $\mathtt{h}_i$ are binding occurrences that can be used in `tac` to refer to a specific hypothesis. Another aspect worth mentioning is keyword `context`. Pattern matching with `context[e]` is successful if `e` occurs as a subexpression of some hypothesis or in the conclusion. In Figure 2, we use `context` to automate case analysis on equality tests on identifiers and natural numbers, as shown below

```
 [ |- context[eq_id_dec ?a ?b] ] =>
          destruct (eq_id_dec a b) ; subst ; try congruence
```

Tactic `destruct` performs case analysis on a term, `subst` searchs the context for a hypothesis of the form `x = e` or `e = x`, where `x` is a variable and `e` is an expression, and replaces all occurrences of `x` by `e`. Tactic `congruence` is a decision procedure for equalities with uninterpreted functions and data type constructors [6].

```
Ltac s :=
  match goal with
    | [ H : _ /\ _ |- _] => destruct H
    | [ H : _ \/ _ |- _] => destruct H
    | [ |- context[eq_id_dec ?a ?b] ] =>
             destruct (eq_id_dec a b) ; subst ; try congruence
    | [ |- context[eq_nat_dec ?a ?b] ] =>
             destruct (eq_nat_dec a b) ; subst ; try congruence
    | [ x : (id * ty)%type |- _ ] =>
             let t := fresh "t" in destruct x as [x t]
    | [ H : (_,_) = (_,_) |- _] => inverts* H
    | [ H : Some _ = Some _ |- _] => inverts* H
    | [ H : Some _ = None |- _] => congruence
    | [ H : None = Some _ |- _] => congruence
    | [ |- _ /\ _] => split
    | [ H : ex _ |- _] => destruct H
  end.

Ltac mysimp := repeat (simpl; s) ; simpl; auto with arith.
```

Fig. 2. Main proof state simplifier tactic.

Tactic `inverts* H` generates necessary conditions used to prove `H` and afterwards executes tactic `auto`.[9] Tactic `split` divides a conjunction goal in its constituent parts.

Besides $\mathcal{L}$tac scripts, the main tool used to automate proofs in our development is tactic `auto`. This tactic uses a relatively simple principle: a database of tactics is repeatedly applied to the initial goal, and then to all generated subgoals, until all goals are solved or a depth limit is reached.[10] Databases to be used — called *hint databases* — can be specified by command `Hint`, which allows declaration of which theorems are part of a certain hint database. The general form of this command is:

```
Hint Resolve thm1 thm2 ... thmn : db.
```

where $\texttt{thm}_i$ are defined lemmas or theorems and `db` is the database name to be used. When calling `auto` a hint database can be specified, using keyword `with`. In Figure 2, `auto` is used with database `arith` of basic Peano arithmetic properties. If no database name is specified, theorems are declared to be part of hint database `core`. Proof obligations for termination are filled using lemmas 7, 8 e 9 that are included in hint databases. Failures of unification, for a given list of constraints $\mathbb{C}$, is represented by `UnifyFailure` and proof obligations related to failures are also handled by `auto`, thanks to the inclusion of `UnifyFailure` constructors as `auto` hints using command

```
Hint Constructors UnifyFailure.
```

6 Related Work

Formalization of unification algorithms has been the subject of several research works [8–11].

In Paulson's work [8] the representation of terms, built by using a binary operator, uses equivalence classes of finite lists where order and multiplicity of elements is considered irrelevant, deviating from simple textbook unification algorithms ([13, 12]).

Bove's formalization of unification [9] starts from a Haskell implementation and describes how to convert it into a term that can be executed in type theory by acquiring an extra termination argument (a proof of termination for the actual input) and a proof obligation (that all possible inputs satisfy this termination argument). This extra termination argument is an inductive type whose constructors and indices represent the call graph of the defined unification function. Bove's technique can be seen as an specific implementation of the technique for general recursion based on well founded relations [26], which is the one implemented on Coq's standard library, used in our implementation. Also, Bove presents soundness and completeness proofs for its implementation together with

[9] This tactic is defined on a tactic library developed by Arthur Charguraud [25].

[10] The default depth limit used by `auto` is 5.

the function definition (as occurs with our implementation) as well as by providing theorems separated from the actual definitions. She argues that the first formalization avoids code duplication since soundness and completeness proofs follow the same recursive structure of the unification function. Bove's implementation is given in Alf, a dependently typed programming language developed at Chalmers that is currently unsupported.

McBride [10] develops a unification function that is structurally recursive on the number of non-unified variables on terms being unified. The idea of its termination argument is that at each step the unification algorithm gets rid of one unresolved variable from terms, a property that is carefully represented with dependent types. Soundness and completeness proofs are given as separate theorems in a technical report [27]. McBride's implementation is done on `OLEG`, a dependently typed programming language that is nowadays also unsupported.

Kothari [11] describes an implementation of a unification function in Coq and proves some properties of most general unifiers. Such properties are used to postulate that unification function does produce most general unifiers on some formalizations of type inference algorithms in type theory [28]. Kothari's implementation does not use any kind of scripted proof automation and it uses the experimental command `Function` in order to generate an induction principle from its unification function structure. He uses this induction principle to prove properties of the defined unification function.

Avelar et al.'s proof of completeness [29] is not focused on the proof that the unifier S of types $\overline{\tau}$, returned by the unification algorithm, is the least of all existing unifiers of $\overline{\tau}$. It involves instead properties that specify: i) $dom(S) \subseteq \mathrm{FV}(\overline{\tau})$, ii) the contra-domain of S is a subset of $\mathrm{FV}(\overline{\tau}) - dom(S)$, and iii) if the unification algorithm fails then there is no unifier. The proofs involve a quite large piece of code, and the program does not follow simple textbook unification algorithms. The proofs are based instead on concepts like the first position of conflict between terms (types) and on resolution of conflicts. More recent work of Avelar et al. [30] extends the previous formalization by the description of a more elaborate and efficient first-order unification algorithm. The described algorithm navigates the tree structure of the two terms being unified in such a way that, if the two terms are not unifiable then, after the difference at the first position of conflict between the terms is eliminated through a substitution, the search of a possible next position of conflict is computed through application of auxiliary functions starting from the previous position.

7 Conclusion

We have given a complete formalization of termination, soundness and completeness of a type unification algorithm in the Coq proof assistant. To the best of our knowledge, the proposed formalization is the first to follow the structure of termination proofs presented in classical textbooks on type systems [13, 12]. Soundness and completeness proofs of unification are coupled with the algorithm definition and are filled by scripted proof tactics using previously proved lemmas.

The developed formalization has 610 lines of code and around 94 lines of comments. The formalization is composed of 31 lemmas and theorems, 34 type and function definitions and 2 inductive types. Most of the implementation effort has been done on proving termination, which takes 293 lines of our code, expressed in 21 theorems. Compared with Kothari's implementation, that is written in more than 1000 lines, our code is more compact.

We intend to use this formalization to develop a complete type inference algorithm for Haskell in the Coq proof assistant. The developed work is available online [19].

References

1. Peyton Jones, S.: Haskell 98 Language and Libraries: the Revised Report. (2003)
2. Milner, R., Tofte, M., Harper, R.: Definition of standard ML. MIT Press (1990)
3. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3) (1978) 348–375
4. Pottier, F., Rémy, D.: The essence of ML type inference. In Pierce, B.C., ed.: Advanced Topics in Types and Programming Languages. MIT Press (2005) 389–489
5. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1) (1965) 23–41
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)
7. Bove, A., Dybjer, P., Norell, U.: A brief overview of agda — a functional language with dependent types. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics. TPHOLs '09, Berlin, Heidelberg, Springer-Verlag (2009) 73–78
8. Paulson, L.C.: Verifying the unification algorithm in lcf. CoRR **cs.LO/9301101** (1993)
9. Bove, A.: Programming in Martin-Löf type theory: Unification - A non-trivial example (November 1999) Licentiate Thesis of the Department of Computer Science, Chalmers University of Technology.
10. McBride, C.: First-order unification by structural recursion. J. Funct. Program. **13**(6) (2003) 1061–1075
11. Kothari, S., Caldwell, J.: A machine checked model of idempotent mgu axioms for lists of equational constraints. In Fernandez, M., ed.: Proceedings 24th International Workshop on Unification. Volume 42 of EPTCS. (2010) 24–38
12. Mitchell, J.C.: Foundations of Programming Languages. MIT Press, Cambridge, MA, USA (1996)
13. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)
14. McBride, C., McKinna, J.: The view from the left. J. Funct. Program. **14**(1) (2004) 69–111
15. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7) (2009) 107–115
16. Barthe, G., Dufay, G., Jakubiec, L., de Sousa, S.M.: A formal correspondence between offensive and defensive javacard virtual machines. In Cortesi, A., ed.: VMCAI. Volume 2294 of Lecture Notes in Computer Science., Springer (2002) 32–45

17. Gonthier, G.: The four colour theorem: Engineering of a formal proof. In Kapur, D., ed.: ASCM. Volume 5081 of Lecture Notes in Computer Science., Springer (2007) 333
18. Gonthier, G.: Engineering mathematics: the odd order theorem proof. In Giacobazzi, R., Cousot, R., eds.: POPL, ACM (2013) 1–2
19. Ribeiro, R., et al.: A mechanized textbook proof of a type unification algorithm — on-line repository. https://github.com/rodrigogribeiro/unification (2015)
20. Sørensen, M., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism. Number v. 10 in Studies in Logic and the Foundations of Mathematics. Elsevier (2006)
21. Chlipala, A.: Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant. MIT Press (2013)
22. de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae (Proceedings) **75**(5) (1972) 381 – 392
23. Charguéraud, A.: The locally nameless representation. J. Autom. Reasoning **49**(3) (2012) 363–408
24. Coq Developement Team: Coq Proof Assistant — Reference Manual. `http://coq.inria.fr/distrib/current/refman/ /` (2014)
25. Pierce, B.C., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjoberg, V., Yorgey, B.: Software Foundations. Electronic textbook (2015)
26. Nordström, B.: Terminating general recursion. BIT Numerical Mathematics **28**(3) (1988) 605–619
27. McBride, C.: First-order unification by structural recursion — correctness proof
28. Naraschewski, W., Nipkow, T.: Type inference verified: Algorithm w in isabelle/hol. J. Autom. Reason. **23**(3) (November 1999) 299–318
29. Avelar, A.B., de Moura, F.L.C., Galdino, A.L., Ayala-Rincón, M.: Verification of the completeness of unification algorithms à la robinson. In Dawar, A., de Queiroz, R.J.G.B., eds.: Logic, Language, Information and Computation, 17th International Workshop, WoLLIC 2010, Brasilia, Brazil, July 6-9, 2010. Proceedings. Volume 6188 of Lecture Notes in Computer Science., Springer (2010) 110–124
30. Avelar, A.B., Galdino, A.L., de Moura, F.L.C., Ayala-Rincón, M.: First-order unification in the PVS proof assistant. Logic Journal of the IGPL **22**(5) (2014) 758–789

Time Performance Formal Evaluation of Complex Systems

Valdivino Alexandre de Santiago Júnior[1] and Sofiène Tahar[2]

[1] Instituto Nacional de Pesquisas Espaciais (INPE)
Av. dos Astronautas, 1758, São José dos Campos, São Paulo, SP, Brazil.
valdivino.santiago@inpe.br
[2] Concordia University
1455 De Maisonneuve Blvd. West, Montreal, QC, Canada.
tahar@ece.concordia.ca

Abstract. Formal verification methods, such as Model Checking, have been used for addressing performance/dependability analysis of systems. Such formal methods have several advantages over traditional techniques aiming at performance/dependability analysis such as the use of a single computational technique for evaluation of any measure and all complex numerical computation steps are hidden to the user. This paper reports on the use of Probabilistic Model Checking for time performance evaluation of complex systems. We propose an approach, TPerP, that allows a professional to clearly address time performance analysis based on Continuous-Time Markov Chain (CTMC). Our approach takes into consideration several types of delay analyzes. We applied it to a balloon-borne high energy astrophysics scientific experiment where we dealt with CTMCs that had around 30 million reachable states and 75 million transitions, and we concluded that the current definition used in the balloon system is inadequate in terms of performance.

1 Introduction

Studies about performance evaluation of systems date back to the early 1900s where single queues, Markov Chains, networks of queues and Stochastic Petri Nets have been used for this purpose. Particularly, Markov Chains have been applied to performance assessment since around 1950 [1].

Performance evaluation is thus a mature field. However, formal verification methods, such as Model Checking and Theorem Proving, have also been used for addressing performance/dependability analysis of systems. Such formal methods have several advantages over traditional techniques (e.g. simulation) aiming at performance/dependability analysis. For instance, temporal logic offers a high degree of expressiveness and flexibility where most standard performance measures (e.g. transient probabilities, long-run likelihoods) can be easily expressed. Moreover, it is possible to specify complex measures in a succinct way by nesting temporal logic formulas [2].

Model Checking is a fully algorithmic approach towards performance evaluation where a single computational technique is used for assessment of any possible measure, and its time and space complexity is attractive. In the worst case scenario, the time complexity is linear in the size of the measure specification (logic formula), and polynomial (order 2 or 3, at most) in the number of states of the stochastic process. Regarding space complexity, in the worst case, it is quadratic considering the number of states of the stochastic process [2, 3]. Not less important, especially for practitioners, using Model Checking for performance evaluation is interesting because all algorithmic/implementation details, all detailed and complex numerical computation steps are hidden to the user.

In this paper, we report on the use of Probabilistic Model Checking for time performance evaluation of complex systems. We organized the activities accomplished in this work on an approach, **Time Performance Evaluation via Probabilistic Model Checking** (TPerP), that allows a professional to clearly address time performance analysis based on Continuous-Time Markov Chain (CTMC) and Probabilistic Model Checking [3–6]. Even though TPerP is based on standard steps defined for Model Checking, it takes into consideration several types of delay analyzes and provides directives so that industry professionals may use it for the development of real and complex systems/software. We applied it to a balloon-borne high energy astrophysics experiment under development at the *Instituto Nacional de Pesquisas Espaciais* (INPE) in Brazil. We dealt with CTMCs that had around 30 million reachable states and 75 million transitions and thus Probabilistic Model Checking confirmed to be a suitable solution for the performance analysis of complex systems. We concluded that the current definition used in the balloon system is inadequate in terms of performance.

This paper is structured as follows. Section 2 presents our approach, TPerP. Section 3 shows the characterization of the problem providing details about the case study. Results and considerations by applying TPerP to the space system are in Section 4. Section 5 presents related work. In Section 6, conclusions and future directions are mentioned.

2 The TPerP Approach

TPerP takes advantage of Probabilistic Model Checking and CTMC to assist in time performance analysis of complex system/software projects. The TPerP approach is shown in Figure 1.

The first step that should be accomplished is to define the parameters to be addressed. These parameters are variables that affect the time performance of the system. TPerP aims at finding the most suitable/optimal values of such parameters. Some examples of these variables are the amount of packages a buffer of a hardware device must store, the size of a packet to be sent from one equipment to another, and the number of sensor measurements to be sent to a central management computer. All these parameters are considered taking into account the time performance perspective.

In computer networking or telecommunications, there are various types of delays (d_i). TPerP considers the following ones: (i) Propagation Delay: $PD = l/s$. It is the ratio between the link length (l) and the propagation speed (s)

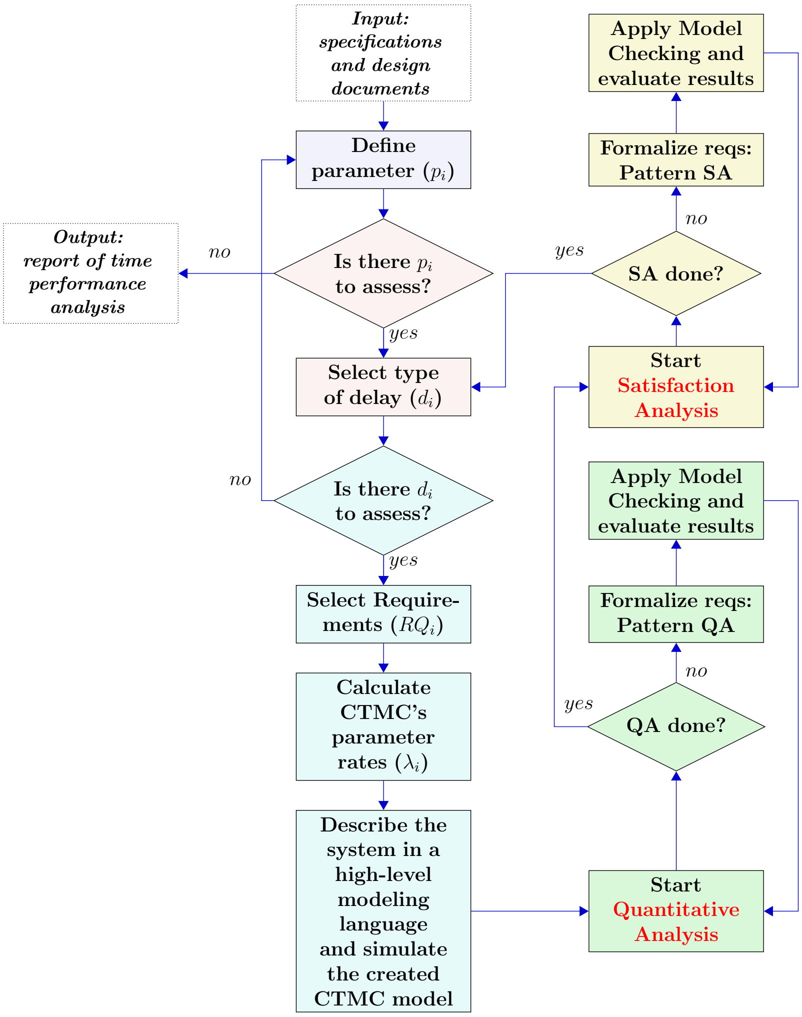

Input: specifications and design documents
Define parameter (p_i)
Is there p_i to assess?
no
Output: report of time performance analysis
yes
Select type of delay (d_i)
Is there d_i to assess?
no
yes
Select Requirements (RQ_i)
Calculate CTMC's parameter rates (λ_i)
Describe the system in a high-level modeling language and simulate the created CTMC model
Start Quantitative Analysis
QA done?
yes
no
Formalize reqs: Pattern QA
Apply Model Checking and evaluate results
Start Satisfaction Analysis
SA done?
yes
no
Formalize reqs: Pattern SA
Apply Model Checking and evaluate results

Fig. 1. The TPerP approach. Caption: QA = Quantitative Analysis; SA = Satisfaction Analysis

over the specific medium. In wireless communication, the speed is the light's propagation speed; (ii) Transmission Delay: $TxD = PDU\_size/bit\_rate$. It is the amount of time from the beginning until the end of a message transmission. It depends on the size of the Protocol Data Unit (PDU) and the speed set for the physical transmission medium; and (iii) Time for PDU Generation: TPG. This is the time needed to create/mount a PDU such as as packet or a frame. It basically depends on the requirements for generating system's data.

It is possible that one d_i can be indeed a combination of two or all of the delays defined above. In addition to these, other delays such as queue processing delays can be addressed. Once a type of delay is chosen, TPerP suggests to select the quantitative requirements based on the system/software specifications.

The next step is to calculate CTMC's parameter rates, λ_i. There should be defined as many λ_i as the amount of relevant flows of data transmission between communicating units. Besides, the calculation of λ_i should be done considering the piece of information that is transmitted per unit of time resolution. Although other ways to calculate the parameter rates are possible, we would like to emphasize two of them:

a) Local PDU influence.

$$\lambda_i = \frac{bit\_rate}{PDU\_size \times time\_res}. \tag{1}$$

In this case, λ_i is calculated simply due to the size of the PDU of the local flow of data transmission;

b) Diverse processing delays and encapsulation influence.

$$\lambda_i = \frac{bit\_rate}{\left(PDU\_size + \left(\sum_j pd_j + \sum_k en_k\right) \times bit\_rate\right) \times time\_res}. \tag{2}$$

Here, $\sum_j pd_j$ and $\sum_k en_k$ represent, respectively, the influence of delays such as queue processing and the impact of other sizes mainly related to the encapsulation features of network architecture models such as Open Systems Interconnection (OSI) and The Consultative Committee for Space Data Systems (CCSDS). Note that in order to use Equation 2, it is necessary to consider at least one influence (processing delays or encapsulation) or both.

Let us assume that a certain flow of data transmission between two computing devices has the following characteristics: PDU size = 65,535 Bytes, bit rate = 1 Gbps, queuing and end system delays (encoding, decoding) = 50 ms. Let us also assume that a encoding system is used where each Byte of data is assigned a 10-bit code like the 8b/10b encoding used in Gigabit Ethernet. In addition, let us consider that a certain requirement, RQ_i, demands that the system must meet a time bound in the range of milliseconds. Thus, the parameter rate is in accordance with Equation 2:

$$\lambda = \frac{10^9}{(65535 \times 10 + 0.05 \times 10^9) \times 1000} = 0.019741. \tag{3}$$

In Equation 3, λ means that, for each millisecond, 0.019741 of a PDU is received by the destination device.

Describing the system in a high-level language supported by a Probabilistic Model Checker such as PRISM [4] is the next step. TPerP suggests the traditional procedure where CTMCs are not obtained in a straightforward way, but rather generated automatically from higher-level descriptions in languages such as stochastic Petri nets, stochastic process algebras [5], or state-based languages [4]. This description is read into a Stochastic/Probabilistic Model Checker which then automatically creates a CTMC based on it. Besides, it is very important to simulate the CTMC model thoroughly in order to avoid an excessive number of false positive counterexamples. The simulation should be carried out by means of the features (outputs) provided by Model Checkers, for example, checking if there are transitions between states of the CTMC model consistent with the expected behavior of the system, and according to the defined parameter rates.

TPerP provides some guidelines to develop the description of the system. In Model Checkers that allow synchronization, our approach suggests using it as much as possible so that modules can make transitions simultaneously. Using auxiliary variables, such as boolean ones, to indicate the end of certain processing is not advisable. Such variables increase the state space and, for the purpose of time performance analysis modeling, they may be replaced by synchronization. For instance, in PRISM, some parts of the modules may be as shown below.

```
module device1
...
[action1] var1 = max_var1 -> rate1 : (var1' = 0);
...

module device2
...
[action1] var2 < max_var2 -> 1 : (var2' = var2 + 1);
...
```

We can see that *action1* synchronizes *device1* and *device2* avoiding the use of a boolean variable to state that a certain PDU is ready to be transmitted from one device to another.

2.1 Quantitative Analysis

Probabilistic Property Specification Templates (ProProST) is a specification patterns system for probabilistic properties as they are used for quality requirements [7]. ProProST complements and extends existing patterns systems [8, 9], in that it allows to specify probabilistic properties as they are required to formulate quality properties. ProProST provides a solution in the form of a formal specification template for Continuous Stochastic Logic (CSL) [3].

Specification patterns systems are very important for Model Checking real world applications since they provide templates/guidelines so that professionals can use them in order to formalize their requirements/properties. Based on this fact, TPerP directs the practitioner to use the *Probabilistic Until* pattern of ProPoST [7] for CSL:

$$\mathcal{P}_{\bowtie p}[\Phi_1 \ \ \mathcal{U}^{[t_1,t_2]} \ \ \Phi_2]. \tag{4}$$

In TPerP, we call this Pattern for Quantitative Analysis (*Pattern QA* in Figure 1) and it is presented in Equation 5:

$$\overline{(\alpha, \mathcal{P}_{=?}[pdu\_notsent\ \ \mathcal{U}^{\leq T}\ \ pdu\_received])}. \tag{5}$$

However, there are three remarks if we compare ProProST's formula (Equation 4) with TPerP's formula (Equation 5). First, some Model Checkers such as PRISM allow to specify properties in a quantitative way. In other words, rather than just verifying and answering true or false whether or not a probability is above or below a certain bound, such tools allow to return the actual probability of some behavior of the model under consideration. For time performance evaluation this is very suitable because in many situations we are interested in knowing what is the optimal value of a parameter p_i based on the time constraints defined in the requirements. If we know the exact value of a probability, we are able to respond more adequately this question. TPerP suggests using the $\mathcal{P}$ operator in this way: $\mathcal{P}_{=?}$. As a result, we will have the value of the probability for the paths starting in the initial state ι.

Naturally, several PDUs may be transmitted from one device to another. Therefore, it is interesting when accomplishing a time performance evaluation to consider a fine-grained analysis, i.e. to take into account α states from where to start the analysis instead of doing it from the initial state ι. Thus, α in TPerP's formula (Equation 5) means precisely to start the analysis from the states that indicate that a PDU_k is ready to be sent from the source device but which has not yet been transmitted to the destination device. Thus, TPerP takes the average values of probabilities for the paths that satisfy the path formula $[pdu\_notsent\ \ \mathcal{U}^{\leq T}\ \ pdu\_received]$, such paths start in α states.

The third remark is about the time interval $[t_1, t_2]$. By default, TPerP instantiates this time interval as $[0, T]$. But it is perfectly possible to change the lower bound from 0 to other real number just assuring that $t_1 \leq t_2$.

Let us consider the following requirement from the automotive industry [10]: *If the system's diagnostic request IRTest is set, then the infrared lamps are turned on after at most 10 seconds.* Thus, a possible physical architecture has an Electronic Control Unit (ECU_{diag}) responsible for ordering the diagnostic request and other unit (ECU_{lamp}) which indeed turns the infrared lamps on. So, communication could take place via one of the protocols used in the automotive industry such as the Controller Area Network (CAN). Hence, the formalization of the requirement is as follows:

$$\overline{(\alpha, \mathcal{P}_{=?}[request\_notsent\_k\ \ \mathcal{U}^{\leq 10}\ \ request\_received - lampson\_k])}. \tag{6}$$

In Equation 6, the state formula $request\_notsent\_k$ means that the kth request (e.g. a CAN frame or a PDU of a CAN-based higher-layer protocol) is ready, in the ECU_{diag}, to be sent to the ECU_{lamp} but has not yet been transmitted. The state formula $request\_received - lampson\_k$ means that not only this kth request has been transmitted and received by ECU_{lamp} but also that the ECU_{lamp} turned the lamps on. The state formulas are usually characterized by means of model variables that indicate the states in which the PDU has not been sent (*notsent*) and that the same PDU has been received (*received*).

After applying Model Checking and evaluating the results, it should be decided whether the quantitative analysis must be ended or not. It is very beneficial that "new" requirements are created by modifying (usually to a larger value) the time constraints defined in the original specifications' requirements. This activity is very useful to find out how inadequate is a particular solution of a system in terms of performance. Therefore, the quantitative analysis should be repeated for each new defined requirement (new time value).

2.2 Satisfaction Analysis

The satisfaction analysis proposed by TPerP is in line with the traditional verification way of properties assessment in Model Checking. This step is particularly suited to realize whether an existing system/software solution is in accordance with system/software time requirements. If a solution does not satisfy a time requirement, thus we can determine which are the necessary modifications in the solution to achieve this goal.

Again the *Probabilistic Until* pattern of ProPoST [7] for CSL is used, starting the analysis from α states as explained in the previous section. In TPerP, we denote this Pattern for Satisfaction Analysis (*Pattern SA* in Figure 1) and it is given by:

$$\exists(\alpha, \mathcal{P}_{\geq bound}[pdu\_notsent\ \ \mathcal{U}^{\leq T}\ \ pdu\_received]). \tag{7}$$

A note should be made on the $\exists$ quantifier in this TPerP's formula (Equation 7). We took advantage of Model Checkers such as PRISM that allow to reason whether there are some paths that satisfy the path formula $[pdu\_notsent\ \ \mathcal{U}^{\leq T}\ \ pdu\_received]$, such paths start in α states, with a probability greater than or equal to a probability *bound*.

It is important to stress that, by using $\exists$, we are not going into the formal details of the qualitative fragment of Probabilistic Computation Tree Logic (PCTL) or CSL. Path quantifiers, $\forall$ and $\exists$, present in the syntax of CTL were replaced by the probabilistic operator $\mathcal{P}_{bound}$ in PCTL and consequently in CSL. We just relied on the benefits of some tools that enable us to reason about a fine-grained time performance analysis.

Let us consider again the real-time requirement from the automotive industry [10] presented in Section 2.1. Its formalization in accordance with *Pattern SA* can be:

$$\exists(\alpha, \mathcal{P}_{\geq 0.98}[request\_notsent\_k\ \ \mathcal{U}^{\leq 10}\ \ request\_received - lampson\_k]). \tag{8}$$

Similarly to the quantitative analysis, it should be decided whether the satisfaction analysis continues or not, in accordance with the same reasoning of creating new requirements by varying time. Likewise, the satisfaction analysis should be repeated for each new defined requirement.

3 Case Study: Characterization of the Problem

protoMIRAX is a hard X-ray imaging telescope [11] which has been developed at the *Instituto Nacional de Pesquisas Espaciais* (INPE) in Brazil. This scientific experiment will be launched by a balloon and it will operate between 40 to 42 km of altitude. The main objective of protoMIRAX is to carry out imaging spectroscopy of selected bright X-ray sources to demonstrate the performance of a prototype of the MIRAX satellite's instrument. A very simplified view of the protoMIRAX's physical architecture is shown in Figure 2.

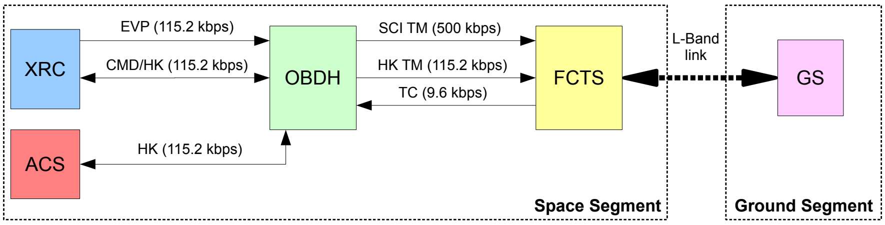


Fig. 2. Simplified physical architecture of the protoMIRAX balloon experiment

The On-Board Data Handling Subsystem (OBDH) is responsible for acquiring, formatting and transmitting all TeleMetry (TM) data that come from several subsystems (X-ray Camera (XRC), Attitude Control Subsystem (ACS)) of protoMIRAX's space segment to the Ground Station (GS). The OBDH is also responsible for receiving and retransmitting, when necessary, various types of TeleCommands (TCs) sent by the GS to the space segment. For each X-ray photon detected by the XRC, a 6-Byte packet is created encasing the time stamp, $x-y$ position, and energy (pulse height) of the event (X-ray photon). This 6-Byte unit of information is called an *event packet* (EVP) and such event packets are sent every 1 second to the OBDH by means of a 115.2 kbps RS-422 unidirectional serial interface. XRC generates 40 event packets/s.

In order to transmit scientific data TM (SCI TM) to ground, a 500 kbps synchronous channel connects the OBDH with the Flight Control and Telecommunications Subsystem (FCTS). Housekeeping data TM (HK TM) are sent to the GS via a serial RS-232 channel operating at 115.2 kbps.

The total size of TC and TM (focus of this research) packets are variable and defined according to each space mission. The problem that needs to be solved by this research is as follows. As the OBDH continuously stores on-board and sends to the GS all TM data (scientific, housekeeping, etc.) it obtains, what is the most suitable (optimal) size of the scientific data TM packet, with event packets created by the XRC and sent to the OBDH, so that we have the minimum delay of such scientific data stored on-board and the same data that are visualized on the GS in real-time? In previous INPE's balloon scientific instrument, this delay was in the range of hour. A suitable performance analysis was not accomplished for this previous project.

4 Application of TPerP

We applied TPerP to the protoMIRAX instrument. We defined one parameter, p_1, which is the size of the scientific data TM packet. Thus, we would like to know what is the most suitable (optimal) value of p_1. We will denote this optimal value as OP_EVP. XRC generates 40 event packets/s and transmits to the OBDH which formats and generates a scientific data TM packet to be sent to the GS. OP_EVP is a value which ranges from 1 to 108. In other words: (i) Minimum size of TM packet ($OP\_EVP = 1$; Size in Bytes = 274). This implies 1 × 40 event packets. Once the OBDH receives the 40 event packets in each second, it formats, stores the data in its memory, and transmits them straight to the GS; and (ii) Maximum size of TM packet ($OP\_EVP = 108$; Size in Bytes = 26,596). This implies 108 × 40 event packets. In other words, the OBDH waits for the arrival of 108 units of 40 event packets and, after that, the OBDH sends such data to the GS.

4.1 Transmission Delay

We chose two types of delays where the Transmission Delay (TxD) was the first one. Thus, one performance requirement was taken into account: RQ1 - *The delay between the scientific data TM packet stored in the OBDH's computer memory and the same data received by the GS must be, at maximum, 500 milliseconds.*

Four parameter rates were considered according to 4 flows of data transmission, and all such rates were calculated taken into account a resolution in ms: (i) $\lambda_{evp} = EVP\_rate/1000 = 0.04$. This is the XRC's event packets generation rate; (ii) $\lambda_{xrc\_sci} = (bit\_rate_{x-o})/(PDU\_size_x \times 1000) = 0.045714$. This rate relates to the transmission of a unit (40) of event packets from the XRC to the OBDH. Note that it is a local PDU influence rate (Equation 1); (iii) $\lambda_{obdh\_sci} = (bit\_rate_{o-g})/((PDU\_size_o + (qd + od) \times bit\_rate_{o-g}) \times 1000)$. This rate relates to the transmission of scientific data (event packets), after being completely formatted, from the OBDH to the GS. It is a diverse processing delay rate (Equation 2) where qd it is the queue processing delay within the software embedded into the OBDH's computer, and od refers to other delays due to further required processing. Note that the size of the OBDH's PDU (PDU_size_o) is directly proportional to OP_EVP. Hence, for $OP\_EVP = 1$, $\lambda_{obdh\_sci} = 0.003285$, and for $OP\_EVP = 108$, $\lambda_{obdh\_sci} = 0.001378$; and (iv) $\lambda_{all\_hk} = (bit\_rate_{hk-o-g})/((PDU\_size_h + (qd + od) \times bit\_rate_{hk-o-g}) \times 1000) = 0.001525$. This rate relates to the transmission of housekeeping data from several subsystems (XRC, OBDH, ACS) to the OBDH which then sends housekeeping information to the GS. It is also a diverse processing delay rate.

We used PRISM and hence we described our system using its language, and simulated the behavior of the CTMC model. Our models range from 546,530 reachable states and 1,647,070 transitions ($OP\_EVP = 1$) to **29,785,885 reachable states and 75,502,215 transitions** ($OP\_EVP = 108$). After realizing that the CTMC model truly reflects our system, we formalized RQ1 as proposed by TPerP:

$$\overline{(\alpha, \mathcal{P}_{=?}[onboard\_TM\_k \ \mathcal{U}^{\leq 500} \ ground\_TM\_k])}. \tag{9}$$

The state formula $onboard\_TM\_k$ means that the kth scientific data TM packet is formatted and ready, in the OBDH, but has not yet been transmitted to the GS while the state formula $ground\_TM\_k$ means that such kth packet has been transmitted and received by the GS. As explained in Section 2.1, α are the states from where to start the analysis because they represent the situations where PDU_k is ready to be sent from the OBDH but such PDU has not yet been sent and received by the GS.

We did several experiments and created several graphics varying the time in accordance with $0 \leq t \leq$ T, where T = 500 ms. Due to space restrictions we will not show them. The current solution defined for the protoMIRAX system is $OP\_EVP = 1$ (minimum). Analyzing the results of the Model Checking, we noticed that such a solution is not a good option because the average value of the probability is too low (0.3316) when T = 500 ms.

As OP_EVP increases, we could see a significant improvement on the average value of the probability when T = 500 ms. However, using a large value of OP_EVP is not the best solution. When $OP\_EVP = 108$ (maximum), the average value of the probability for T = 500 ms is only 0.4924. Figure 3 shows the mean values of probabilities for T = 500 ms for all possible values of OP_EVP. We perceive that there is a set of optimal values: $12 \leq OP\_EVP \leq 19$. The highest mean probability is due to $OP\_EVP = 15$ (0.6954).

We continued the quantitative analysis in order to find out how unsuitable was the current solution ($OP\_EVP = 1$). Thus, we changed RQ1 and created a new requirement where the time is now 1 hour. We noticed a small improvement but the average value of the probability reaches a limit still too low (0.3581). Importantly, the result of this analysis does not claim that the scientific data will last one hour, or even more, to reach ground (GS). The maximum value of the probability for $OP\_EVP = 1$ is, in fact, 0.8065. However, a low mean value of probability means that, on average, significant delays may occur with greater probability when the operation of the protoMIRAX system.

Regarding the satisfaction analysis, we accomplished it in order to answer this question: given the current characteristics of the protoMIRAX system (packet sizes, communication rates, etc.) is RQ1 satisfied? The previous quantitative analysis has provided an indication of what value, or interval, of OP_EVP would be the most appropriate. Such analysis also suggests that the current solution, $OP\_EVP = 1$, is inappropriate. But nothing was said concerned the satisfaction of RQ1. We formalized RQ1 in accordance with TPerP:

$$\exists(\alpha, \mathcal{P}_{\geq 0.98}[onboard\_TM\_k \ \mathcal{U}^{\leq 500} \ ground\_TM\_k]). \tag{10}$$

In Table 1, we see that RQ1 is NOT satisfied in accordance with the current characteristics of the protoMIRAX experiment. No value of OP_EVP was such that the CTMC model satisfied RQ1 (T = 0.5 s). Note that the current characteristics of the protoMIRAX system (packet sizes, communication rates) only

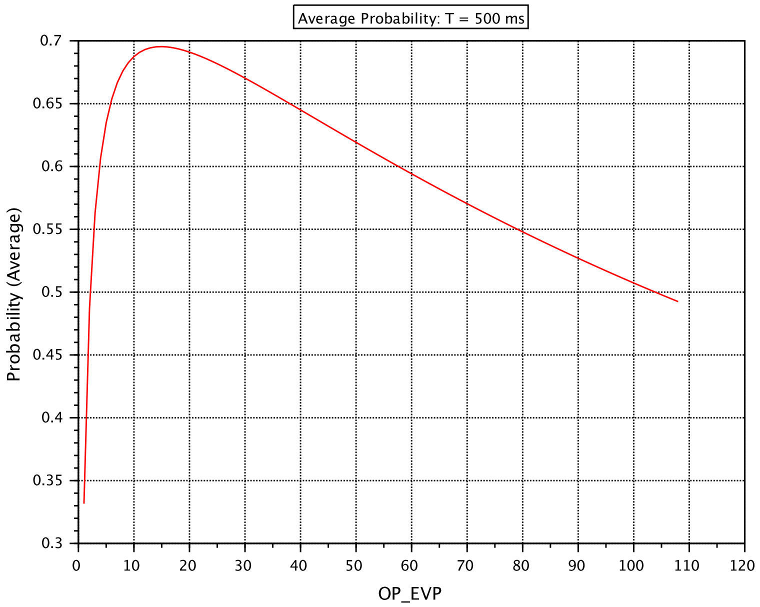


Fig. 3. TxD: Average values of probabilities considering T = 500 ms (all OP_EVP)

begins to fulfill the time performance requirements from T $\geq$ 2 s but, even so, for some values of OP_EVP (10, 13, 15, 17, and 30). It is important to stress that neither for the minimum (1) nor for the maximum (108) value of OP_EVP the requirement is satisfied when T = 2 s. In addition, the minimum value (1) and current solution does not satisfy the requirements even if we consider T = 1 day (86,400 s). This is another result that corroborates the previous conclusion: current solution ($OP\_EVP = 1$) is inadequate.

Table 1. TxD: Satisfaction of time performance requirements

OP_EVP	**Time (seconds)**					
	0.5	**1**	**2**	**5**	**10**	**86,400**
1	✗	✗	✗	✗	✗	✗
10	✗	✗	✓	✓	✓	✓
13	✗	✗	✓	✓	✓	✓
15	✗	✗	✓	✓	✓	✓
17	✗	✗	✓	✓	✓	✓
30	✗	✗	✓	✓	✓	✓
108	✗	✗	✗	✓	✓	✓

4.2 Total Time

For protoMIRAX, the Propagation Delay is neglected due to the small distance of operation of the balloon compared to the light's propagation speed. However, it is interesting to consider the Time for PDU Generation (TPG). For this system, such a time is basically defined as a function of OP_EVP. That is, if $OP\_EVP = 10$ thus 1 scientific data TM packet will be ready every 10 seconds (approximately) to be sent to ground. We considered the Total Time which is based on TPG and TxD: $Total\_Time = TPG + TxD$.

We repeated the process suggested by TPerP for this new delay. The requirement is basically the same as previously proposed but with T = 30 s. Since TPG is at least one second, there is no sense in demanding the system to meet a requirement in the range of milliseconds. The semantics to create the CTMC did not change but the parameter rates did (λ_i calculated with resolution in s): (i) $\lambda_{evp} = EVP\_rate = 40$; (ii) $\lambda_{xrc\_sci} = (bit\_rate_{x-o})/(PDU\_size_x + (OP\_EVP \times bit\_rate_{x-o}))$. Note the influence of the encapsulation feature (Equation 2) presented in the calculation of this rate due to its dependency on OP_EVP; (iii) $\lambda_{obdh\_sci} = (bit\_rate_{o-g})/(PDU\_size_o + (qd + od + OP\_EVP) \times bit\_rate_{o-g})$. In this case, we have both dependencies: processing delays and encapsulation; and (iv) $\lambda_{all\_hk} = (bit\_rate_{hk-o-g})/(PDU\_size_h + (qd + od + OP\_EVP) \times bit\_rate_{hk-o-g})$. Again, diverse processing delays and encapsulation influence were used.

Analyzing the results of the Model Checking where $0 \leq t \leq$ T, T = 30s, we realized that, initially, $OP\_EVP = 1$ had a better performance compared with the other values. This is explained by the lower TPG when $OP\_EVP = 1$. However, it was evident that the average value of the probability when $OP\_EVP = 1$ reaches again a low limit (0.6057). The interval $13 \leq OP\_EVP \leq 17$ is a good option, although $OP\_EVP = 10$ was the value which had the highest average value of probability (0.8103). In Figure 4, we see more clearly the mean values of probabilities considering all values of OP_EVP.

Concerning the satisfaction analysis, the protoMIRAX system also did NOT satisfy the requirement initially proposed (T = 30 s). No value of OP_EVP given in Table 1 was such that the requirement could be satisfied. Increasing T to 60 s, then $OP\_EVP = 10$ and $OP\_EVP = 13$ were the only ones to meet the requirement and thus none of the other values, including the minimum ($OP\_EVP = 1$), satisfied the property for T = 60 s.

4.3 Considerations about the Evaluation conducted via TPerP

Based on the time performance analysis accomplished via TPerP, we can conclude: (i) the current solution adopted in the protoMIRAX system, $OP\_EVP = 1$, is inadequate. This conclusion is valid not only if we consider the Transmission Delay but also the Total Time; (ii) the interval $13 \leq OP\_EVP \leq 17$ is a good alternative to solve this performance issue. As a first option, we suggest $OP\_EVP = 15$ (average value of the interval) to be used in the protoMIRAX system. It means a TM packet with 3,718 Bytes. This value is particularly suited

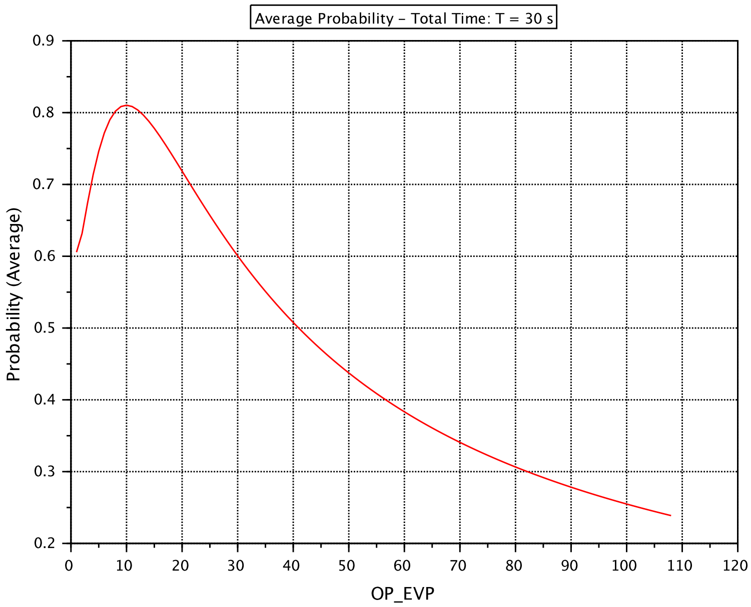


Fig. 4. Total Time: Average values of probabilities considering T = 30 s (all OP_EVP)

if more emphasis is given to a shorter Transmission Delay; and (iii) if we desire a lower Total Time, a second alternative is $OP\_EVP = 10$ (packet with 2,488 Bytes). This value also performed well in the TxD analysis, although it was not as good as $OP\_EVP = 15$. The advantage in using $OP\_EVP = 10$ rather than $OP\_EVP = 15$ would be the fact of having more frequent updating of scientific data visualized on the GS's computers.

We can highlight some points given the results presented by using TPerP for this space system, aiming to apply our approach to other types of systems. For applications that need to store the acquired data and transmit them to another remote system, using a minimum value of a parameter (size of memory buffer, size of a packet) may be, at first, advantageous because, usually, this implies less processing demands (for example, less complex memory management). However, not always this minimum value may be the most appropriate (as we showed in this study), especially considering real-time systems where performance requirements may have greater relevance.

Different types of delays may require different solutions for the optimal values of the parameters that are being evaluated. The decision to choose the most suitable value of the parameters will depend on a priority of the considered delays.

5 Related Work

In this section, we will focus on some relevant studies that use formal verification methods aiming at performance/dependability evaluation of systems.

Probabilistic Model Checking was applied to address usability concerns that involve quantitative aspects of a user interface for a groupware system [5]. There are usability issues that are influenced by the performance of the groupware system rather than by its functional behavior. Our approach allows a more refined analysis considering the size of PDUs to estimate the rates compared with their work. In addition, the CTMC model they developed is very simple (19 states) which raises doubts whether the same results would be achieved for more complex models.

Probabilistic Model Checking was also used to analyze dependability properties of a embedded controller-based system [4]. Properties like the probability of the system shutting itself down, the expected amount of time spent in different classes of states of the model (using reward-based properties), expected number of reboots, were taken into account. The basic difference between TPerP and this work is that we aimed at evaluating the system time performance considering its normal operational behavior, and in [4] the authors aimed to assess dependability-related issues.

In [6], the authors showed how CSL can be used to specify state- and path-based dependability properties of systems being modeled as CTMC. Although dependability was the focus and properties in CSL to reason about probabilities of Quality of Service (QoS) were considered, some time-performance requirements were assessed. This paper is more a proof of concept to show the potential of the recently, at that time, introduced CSL for dependability/performance analysis.

A report on the use of the COrrectness, Modeling and Performance of Aerospace SyStems (COMPASS) toolset for correctness, dependability, and performance analysis of a satellite system-level design was presented in [12]. The greatest motivation behind their research is having a single, integrated, system model that covers several aspects (discrete, real-time, hybrid, probabilistic) rather than using various (tailored) models covering different aspects. The case study is interesting and complex (50 million states) and they accomplished several analyzes. However, the performability evaluation analysis ran out of memory after 9 hours. Moreover, the analysis was conducted when the satellite project was in its Phase B where requirements were not fully detailed. Thus, it is not very clear if they were able to use detailed and defined requirements as we did in our case study and also whether the performance analysis accomplished considers the same type of time perspective and granularity that we carried out.

In a follow-up paper, authors of [12] published another work where they presented the application of the COMPASS toolset to the same project but addressing Phase C of the satellite's system engineering lifecycle [13]. Thus, there were many more design details than in the previous attempt. However, they focused on diagnosability, not performability, analysis which was intractable in the previous work as it needed more computing resources than they had available.

Performance modeling of concurrent live migration operations in cloud computing systems was presented in [14]. The authors used CTMC and Probabilistic Model Checking as we did. They made some assumptions in their model so that they could accomplish the analysis in a shorter time. For instance, migration requests for sender servers are distributed in a uniform way, and thus they could simplify the model for a large number of servers with the ratio of the number of sender and receiver servers. This is clearly an attempt to deal with the state space explosion problem. It is not evident if this uniform assumption is consistent with the real characteristics of such systems.

We can point out the following differences and in some cases advantages of our approach compared with these previous studies: (i) TPerP has well-defined activities for the application of Probabilistic Model Checking for evaluating a specific type of performance measure (delay). It is vital that systematic procedures are proposed so that formal methods can have a wide acceptance in the industrial practice; (ii) we clearly define equations to calculate the parameter rates of the CTMC model considering local PDU influence, queue processing delays, encapsulation influence due to network architecture models, and time resolution; and (iii) with the exception of the studies [12, 13], all remaining papers that we mentioned used very small case studies. We dealt with complex models and so we believe that our approach is suitable for large scale applications.

6 Conclusions

In this paper we report on the use of Probabilistic Model Checking to evaluate time performance of complex systems. We organized the activities that we carried out in TPerP, an approach that analyzes several types of delay and goes towards a wide acceptance of formal methods in practice. Our approach defines clear steps to be followed by professionals by providing guidelines to calculate parameter rates, and suggesting the use of a specification patterns system.

We applied TPerP to a complex space application under development at INPE aiming at finding the optimal/most suitable size of the scientific data TM packet, so that there is a minimum delay of such scientific data stored onboard and the same data that are visualized on the ground. We found that the current definition of the balloon-borne experiment is inadequate and we suggest different sizes for the TM packet: $OP\_EVP = 15$ if we consider a shorter Transmission Delay; or $OP\_EVP = 10$ if the Total Time is the driving factor. CTMC models up to 30 million reachable states and 75 million transitions were analyzed showing the usefulness of our approach.

Future directions include to propose a method for the automatic translation of the system/software behavior into the high-level modeling language of a Probabilistic Model Checker such as PRISM. This step is quite interesting because it can prevent errors in (manual) translating the system solution into the language of the Model Checker. It is also interesting to investigate the use of other ProPoST's patterns and find out the impact on the time performance analysis. Finally, application to other case studies (aerospace domain, automotive industry, etc.) should be addressed to consider the generalization of our approach.

Acknowledgments

This work was supported by a CNPq BSP grant, Institutional Process Number 455097/2013-5, Individual Process Number 170143/2014-7.

References

1. G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains.* Jonh Wiley & Sons, New York, NY, USA, 1998.
2. C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Performance evaluation and model checking join forces. *Commun. ACM*, 53(9):76–85, 2010.
3. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Trans. Software Engineering*, 29(6):524–541, 2003.
4. M. Kwiatkowska, G. Norman, and D. Parker. Controller dependability analysis by probabilistic model checking. *Control Engineering Practice*, 15(11):1427–1434, 2006.
5. M. H. ter Beek, M. Massink, and D. Latella. Towards model checking stochastic aspects of the thinkteam user interface. In *Interactive Systems. Design, Specification, and Verification*, volume 3941 of *LNCS*, pages 39–50. Springer, 2006.
6. B. R. Haverkort, H. Hermanns, and J.-P. Katoen. On the use of model checking techniques for dependability evaluation. In *Proc. IEEE Symp. Reliable Distributed Systems*, pages 228–237. IEEE, 2000.
7. L. Grunske. Specification patterns for probabilistic quality properties. In *Proc. Int. Conf. Software Engineering*, pages 31–40. ACM, 2008.
8. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. Int. Conf. Software Engineering*, pages 411–420. ACM, 1999.
9. S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proc. Int. Conf. Software Engineering*, pages 372–381. ACM, 2005.
10. A. Post and J. Hoenicke. Formalization and analysis of real-time requirements: A feasibility study at BOSCH. In *Verified Software: Theories, Tools, Experiments*, volume 7152 of *LNCS*, pages 225–240. Springer, 2012.
11. J. Braga, F. D'Amico, M. A. C. Avila, A. V. Penacchioni, J. R. Sacahui, V. A. Santiago Júnior, F. Mattiello-Francisco, C. Strauss, and M. A. A. Fialho. The protoMIRAX Hard X-ray Imaging Balloon Experiment. *Astronomy & Astrophysics*, 580:A108, 2015.
12. M.-A. Esteve, J.-P. Katoen, V. Y. Nguyen, B. Postma, and Y. Yushtein. Formal correctness, safety, dependability, and performance analysis of a satellite. In *Proc. Int. Conf. Software Engineering*, pages 1022–1031. IEEE Press, 2012.
13. M. Bozzano, A. Cimatti, J.-P. Katoen, P. Katsaros, K. Mokos, V. Y. Nguyen, T. Noll, B. Postma, and M. Roveri. Spacecraft early design validation using formal methods. *Reliability Engineering & System Safety*, 132:20 – 35, 2014.
14. S. Kikuchi and Y. Matsumoto. Performance modeling of concurrent live migration operations in cloud computing systems using PRISM Probabilistic Model Checker. In *Proc. IEEE Int. Conf. Cloud Computing*, pages 49–56. IEEE, 2011.

Test Case Generation from Natural Language Requirements using CPN Simulation

Bruno Cesar F. Silva[1], Gustavo Carvalho[1], and Augusto Sampaio[1]

[1]Universidade Federal de Pernambuco - Centro de Informática, 50740-560, Brazil
{bcfs,ghpc,acas}@cin.ufpe.br

Abstract. We propose a test generation strategy from natural language (NL) requirements via translation into Colored Petri Nets (CPN), an extension of Petri Nets that supports model structuring and provides a mature theory and powerful tool support. This strategy extends our previous work on the NAT2TEST framework, which involves syntactic and semantic analyses of NL requirements and the generation of Data Flow Reactive Systems (DFRS) as an intermediate representation, from which target formal models can be obtained for the purpose of test case generation. Our contributions include a systematic translation of DFRSs into CPN models, besides a strategy for test generation. We illustrate our overall approach with a running example.

Keywords: test generation, CPN, model simulation

1 Introduction

Since the so-called *software crisis* (in the Sixties), when the term *software engineering* originated, difficulty of understanding the user needs, ambiguous specifications, poorly specified and interpreted requirements are still common issues. In the field of requirements engineering, several studies have been conducted focusing on the use of (semi-)formal methods to specify, model and analyse requirements, besides automatically generating test cases, resulting in greater maturity and in the construction of underlying theories and supporting tools.

According to Myers [14], as formal methods can be used to find errors in requirements such as inconsistency and incompleteness, they play an important role in the earlier discovery of errors and, thus, reducing costs and correction effort. In this light, we have devised a strategy [2] that generates test cases from natural language requirements. In this strategy, the system behaviour is formally represented as a Data-flow Reactive System (DFRS). Later, this model is translated into a target formalism for generating test cases. The purpose of the NAT2TEST tool is to be easily extensible to various target formalisms.

Although the use of CSP [6], SCR [5] and IMR [3] have been explored, each one has advantages and limitations. For instance, test generation from SCR and IMR is based on commercial tools, which is a practical advantage, but, on the other hand, is not formal, as no explicit conformance relations are considered. Differently, the test strategy for CSP is formal and can be proved sound, with an explicit conformance notion. In this case, the FDR tool[1] is used to generate test cases as counterexamples of refinement verifications and, thus, it demands the complete expansion of the underlying label transition system, which can easily lead to state explosion problems.

Therefore, here we explore the use of Colored Petri Nets (CPN) as an alternative extension of the NAT2TEST strategy where simulation of CPNs is used for generating test cases. To simulate a CPN it is not necessary to explore its complete state space first and, thus, we minimise state explosion problems. In other words, we are capable of generating test cases for more complex systems since it is not necessary to enumerate all its states first, but they can be computed dynamically during simulation.

Also, we benefit from the diversity and maturity of CPN tools[2]. For instance, besides simulation we can also perform model checking. Another promising opportunity is to make use of the CPN ML language, a functional programming language based on Standard ML. Furthermore, a testing theory based on CPN can also be entirely formal, although our focus here is on its feasibility. The main contributions of this paper are the following:

- A systematic translation of DFRSs into CPN models;
- A strategy for test case generation from CPN models via simulation;
- An example to illustrate our overall approach.

Section 2 introduces the NAT2TEST strategy and CPNs. Section 3 explains how to generate CPN models from DFRSs. Section 4 describes how to generate test vectors from CPN models via simulation. Section 5 addresses related work, and presents our conclusions and future directions.

2 Background

Here, we provide a brief explanation of the NAT2TEST strategy and CPNs.

2.1 NAT2TEST Strategy

In this section we briefly describe the *NAT2TEST* (NATural language requirements to TEST cases) strategy for generating test cases from requirements written in natural language. It is tailored to generate tests for *Data-Flow Reactive Systems* (DFRS): a class of embedded systems whose inputs and outputs are always available as digital signals. The input signals can be seen as data provided by sensors, whereas the output data are provided to system actuators.

It receives as input system requirements written using the *SysReq-CNL*, a *Controlled Natural Language* (CNL) specially tailored for editing unambiguous requirements of data-flow reactive systems. As output, it produces test cases. This test-generation strategy comprises a number of phases (see Figure 1). The three initial phases are fixed: (1) syntactic analysis, (2) semantic analysis, and (3) DFRS generation; the remaining phases depend on the internal formalism.

[1] http://www.cs.ox.ac.uk/projects/fdr/

[2] http://cpntools.org/

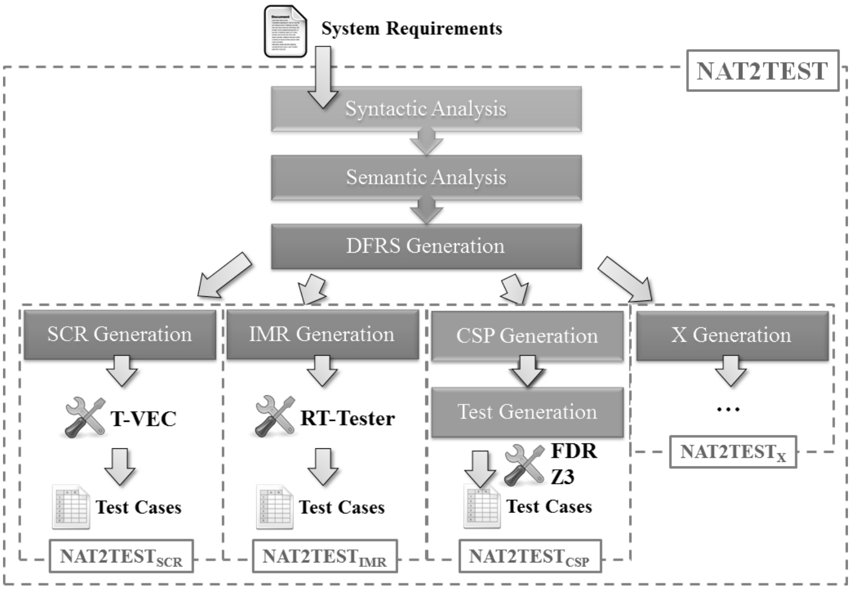


Fig. 1. NAT2TEST strategy

The syntactic analysis phase receives as input the system requirements, and performs two tasks: it verifies whether these requirements are in accordance with the SysReq-CNL grammar, besides generating syntactic trees for each correctly edited requirement. The second phase maps these syntax trees into an informal NL semantic representation. Afterwards, the third phase derives an intermediate formal characterization of the system behaviour (DFRS) from which other formal notations can be derived. The possibility of exploring different formal notations allows analyses from several perspectives, using different languages and tools, besides making the strategy extensible, as discussed in the previous section.

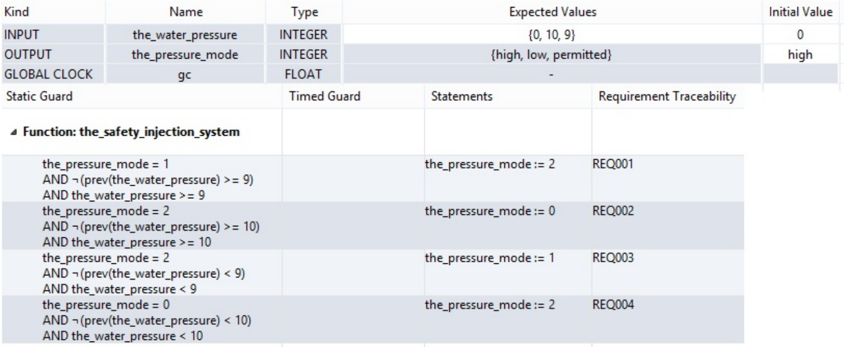
Data-Flow Reactive Systems A DFRS model [4] provides a formal representation of the requirements semantics. It has a symbolic and an expanded representation. Briefly, the symbolic version is a 6-tuple: (I, O, T, $gcvar$, s_0, F). Inputs (I) and outputs (O) are system variables, whereas timers (T) are used to model temporal behaviour. There is a global clock denoted $gcvar$. The element s_0 is the initial state. The last element (F) represents a set of functions, each one describing the behaviour of one system component.

The expanded DFRS comprises a (possibly infinite) set of states, and a transition relation between states. This expanded representation is built by applying the elements of F to the initial state to define *function transitions*, but also letting the time evolve to define *delay transitions*. This expanded representation can be seen as a semantics for symbolic DFRSs.

Figure 2 illustrates a DFRS for a simplified version of the control system for safety injection in a nuclear power plant (NPP) described in [11]. If the water pressure is too low (lower than 9 units), the system injects a coolant into the reactor. In Figure 2, one can note that the corresponding DFRS has one input and one output variable (*the_water_pressure* and *the_pressure_mode*, respectively).

The system behaviour is captured by the function *the_safety_injection_system*. It has four entries. Each entry comprises a 3-tuple: a static guard, a timed guard (here, empty), and the expected system reaction (a list of assignments). The possible pressure modes are *high*, *low*, and *permitted*. While these values are represented in the DFRS as numbers ($high \mapsto 0, low \mapsto 1, permitted \mapsto 2$), they are represented in the corresponding CPN as strings for legibility purposes.

In Figure 2, the first requirement (REQ001) captures the system reaction when the current pressure mode is low ($the\_pressure\_mode = 1$), and the water pressure becomes greater than or equal to 9. As in this example the system reaction does not depend upon the time elapsed, we only have static guards. When the situation previously described happens, the pressure mode changes to permitted ($the\_pressure\_mode := 2$). Similarly, the other requirements describe when the pressure mode changes to (from) low, permitted, and high.



Kind	Name	Type	Expected Values	Initial Value
INPUT	the_water_pressure	INTEGER	{0, 10, 9}	0
OUTPUT	the_pressure_mode	INTEGER	{high, low, permitted}	high
GLOBAL CLOCK	gc	FLOAT	-	

Static Guard	Timed Guard	Statements	Requirement Traceability
Function: the_safety_injection_system			
the_pressure_mode = 1 AND ¬(prev(the_water_pressure) >= 9) AND the_water_pressure >= 9		the_pressure_mode := 2	REQ001
the_pressure_mode = 2 AND ¬(prev(the_water_pressure) >= 10) AND the_water_pressure >= 10		the_pressure_mode := 0	REQ002
the_pressure_mode = 2 AND ¬(prev(the_water_pressure) < 9) AND the_water_pressure < 9		the_pressure_mode := 1	REQ003
the_pressure_mode = 0 AND ¬(prev(the_water_pressure) < 10) AND the_water_pressure < 10		the_pressure_mode := 2	REQ004

Fig. 2. An example of DFRS (variables, types and functions)

2.2 Colored Petri Nets

Colored Petri Nets (CPN) are an extension of high-level Petri nets – a graphical modelling language with formally defined syntax and semantics. As described in [8], the formal definition of a CPN is given by a 9-tuple (Σ, P, T, A, N, C, G, E, I), satisfying the following properties:

- Σ is a finite set of non-empty *color sets* (types).

- P is the finite set of all *places*.
- T is the finite set of all *transitions*.
- A is the finite set of all *arcs* such that $P \cap T = P \cap A = T \cap A = \emptyset$
- N is the *node* function such that $N : A \rightarrow P \times T \cup T \times P$
- C is the *color* function such that $C : P \rightarrow \Sigma$
- G is the *guard* expression function such that $G : T \rightarrow Expr$, and $\forall t \in T : [Type(G(t)) = Bool \wedge Type(Var(G(t))) \subseteq \Sigma]$
- E is the *arc expression* function such that $E : A \rightarrow Expr$, and $\forall a \in A : [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$, where $p(a)$ is the place of $N(a)$;
- I is the *initialization* function such that $I : P \rightarrow Expr$, and $\forall p \in P : [Type(I(p)) = C(p)_{MS}]$

We denote by $Expr$ the set of expressions provided by the inscription language (CPN ML), and by $Type(e)$ the type of an expression $e \in Expr$, i.e., the type of $Var(e)$ (the values obtained when evaluating e). The subscript MS denotes the multiset of the associated place.

Informally, a CPN model expresses both the states and the events of a system. Events are represented by transitions (drawn as rectangles) and the states are represented by places (drawn as ellipses or circles). These two kinds of nodes are connected by arcs that indicate the direction in which tokens (data values) are moved. Arcs are used to connect places with transitions, and vice-versa, but never two places or two transitions. Places hold collections of tokens and, thus, represent local states (markings). The global state (or global marking) of a model is represented by the distribution of tokens throughout its places. The places also have an initial marking representing the initial local state. Figure 3 shows part of the CPN obtained for the simplified NPP (explained in Section 3).

As described in [15], a transition typically represents an event that occurs in the model. When this event occurs, the transition is said to fire. When a transition is enabled it means that it is ready to fire. When a transition fires, it is able to remove tokens from its input places and to produce new tokens on its output places. The inscriptions in the arcs leading to a transition define the quantity of tokens needed for enabling the transition. Moreover, qualitative constraints can be specified for the tokens from the input places. This can be done by the use of pattern matching or guards in the transition.

A guard is a predicate that must evaluate to true for enabling the transition. For a given transition, if it is possible to select a set of tokens from the input places while satisfying these constraints, this transition is enabled. If another transition also depends on some of the same tokens for being enabled, the transitions are said to be in conflict. This property is part of what makes (Colored) Petri Nets a very strong tool for modelling distributed and concurrent systems with shared resources, parallel processes and other characteristics that come with this type of systems.

The graphical definition of CPNs is supplemented by declarations of functions, operations, constants, variables, color sets and priorities; all of them in a functional programming language (CPN ML [9]): an extension of the more commonly known Standard ML [13]. Therefore, CPN models differ from traditional Petri Nets by the fact that the tokens have types (*color sets*) and values.

The CPN modelling language also supports the specification of hierarchically structured models as a collection of connected modules (or pages). This makes it possible to work with different levels of abstraction. A module is itself a CPN model (possibly hierarchical too). Structuring is performed through two mechanisms: fusion places or substitution transitions. Here, we use the former. A fusion place is a set containing multiple places that may be found in different modules. This allows interaction to traverse boundaries of the modules in the model.

CPN also allows the specification of time-based behaviour [10]. This is accomplished adding time stamps to tokens. In this way, delays are expressed as timing annotations in arcs and transitions. Although the DFRS is able to deal with discrete and continuous time representation, we are not dealing with time in our CPN representation yet.

3 CPN Model Generation

Intuitively, the behaviour of a DFRS is encoded as transitions that modify the corresponding output variables, which are modelled as CPN places. Therefore, the target of these transitions are these output places, and their source are auxiliary places that provide to the transitions the system inputs. The generation of CPN models from DFRSs is accomplished in three steps.

In the first step, input variables and their types are translated to CPN variables and color sets, respectively. In second step, output variables are mapped to places along with the corresponding color set. In the last step, the DFRS functions are represented in the CPN model as transitions. In what follows, we detail these steps. To illustrate the relevant concepts, here and in the next section, we consider the simplified NPP briefly described in Section 2.1.

Representing input variables For each input variable of the DFRS, a CPN variable is created along with the corresponding color set (type). Sometimes, the system reaction depends upon the current, but also the previous input value. In such cases, besides creating a variable to represent the system input, we also create another variable to store the previous input value. For instance, considering the simplified NPP, we create two variables to represent the water pressure: *wp* (*water pressure*) and *pwp* (*previous water pressure*), whose type (color set) is *INTWP* – an integer ranging from 0 to 11.

Representing output variables In this case, they are represented by places to denote the system state, along with variable for transmitting the previous and current values, similarly to input variables. Color sets are also created to represent the type of the output variables. When transitions that modify an output variable are fired, it leads to the corresponding place changing its marking and, thus, the value of the corresponding CPN variable accordingly. A place has as initial marking the initial value of the corresponding output variable.

Considering the simplified NPP example, we have a single output variable that represents the pressure mode. There are three possible modes: *high*, *low*, and *permitted*. Therefore, we create a place (*Pressure Mode*), a variable (*pm* – *pressure mode*), and a color set (*PRESSURE* – an enumeration comprising these three values). This place is reached, for instance, when the transition *REQ001* is fired (see Figure 3). As explained in what follows, CPN transitions are created to represent the system requirements. The arc leading to the place *Pressure Mode* has as inscription *permitted*, since the requirement REQ001 describes the situation when the pressure mode becomes permitted.

Representing functions For each entry of a DFRS function, a page is created to represent the behaviour of this entry. It simplifies visual inspection of the model, since each page details just one possible system reaction. For easier traceability of CPN pages to the requirements, the page is named after the requirement related to the considered entry.

Each page comprises some places and one transition. The transition (named after the same requirement) represents a possible system reaction – how the output variables should be updated. Therefore, this transition has outgoing arcs to one or more places (the ones that represent output variables) that it should modify when fired. Besides the places that represent output variables, there are four other places (*Input*, *Next Input*, *Next TO*, and *Test Oracles*) that are auxiliary elements used for generating test cases (later explained in Section 4.1).

To give a concrete example, consider the simplified NPP and the entry related to *REQ001*: $(the\_pressure\_mode = 1 \land \neg(prev(the\_water\_pressure) >= 9) \land the\_water\_pressure >= 9, \emptyset, the\_pressure\_mode := 2)$. It means that when the pressure mode is 1 (*low*), and the water pressure is greater than or equal to 9, but it was not greater than or equal to 9 in the previous state (*prev*), the system shall change the pressure mode to 2 (*permitted*). We note that, as previously said, string enumerations are represent in DFRSs as numbers, whereas we use the original strings in the CPN for legibility purposes.

For this entry, we create the page *REQ001*. In this page, there is a transition *REQ001*. To be enabled, the guard associated to this transition needs to evaluate to true. This guard ($[not(pwp >= 9), (wp >= 9), (pm = low), (n = i)]$) is a direct translation of the guards of the corresponding function entry. The variable *pwp* stands for the previous value of the water pressure. The last element of this guard ($n = i$) is required to ensure the order in which this transition processes the system inputs (later explained in Section 4.1).

Besides satisfying its guard, a transition may also need to know the current and previous values of the system inputs, along with the value of the system outputs. Note that the aforementioned guard also depends upon the value of a system output – the pressure mode (*pm*). These values (tokens) are sent to the transition *REQ001* by the arcs emanating from the places *Input* and *Pressure Mode*. When this guard evaluates to true, and the required tokens are available,

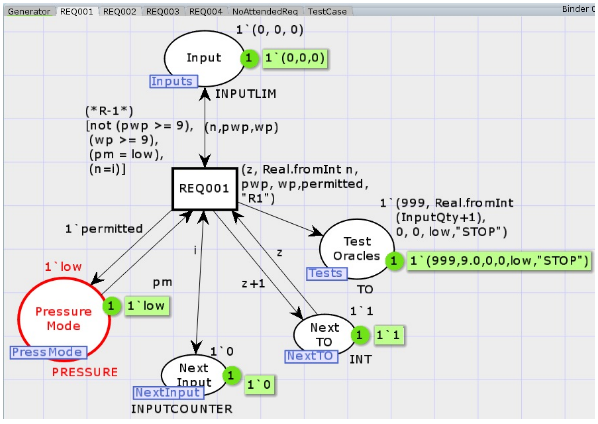


Fig. 3. Page created to represent the requirement REQ001

the transition fires and assigns permitted to the pressure mode. As previously said, for every entry of each DFRS function, a page similar to this one is created.

4 Generating Test Vectors Based on CPN Simulation

In order to generate test cases via CPN simulation, it is necessary to complement the model generated in the previous section with some auxiliary elements. This is the subject of Section 4.1. Section 4.2 presents how test vectors are generated via simulation of a CPN.

4.1 Auxiliary Components

After constructing a CPN to represent a DFRS (see Section 3), we now create auxiliary components, which are used to generate test cases. These components are always the same, but parametrized by the number of inputs and outputs. Considering the constant *InputQty*, which is defined beforehand to limit the number of generated inputs, the following auxiliary elements are created.

Generating entries A page (*Generator*) is created to generate input values randomly. When fired, the transition *Generate Inputs* generates the system inputs, and transmits them, along with an index *i*, to the place *Input* (see Figure 4). This index is used to ensure the order the generated inputs are consumed by the transitions that represent the system behaviour. This index is initially 0, and is incremented with the aid of the place *Next*. We note that the generated inputs are also an input to the transition *Generate Inputs*. It is necessary because when generating the next inputs, the token sent to *Inputs* might comprise the newly generated values, but also the previous values of the inputs. When the number

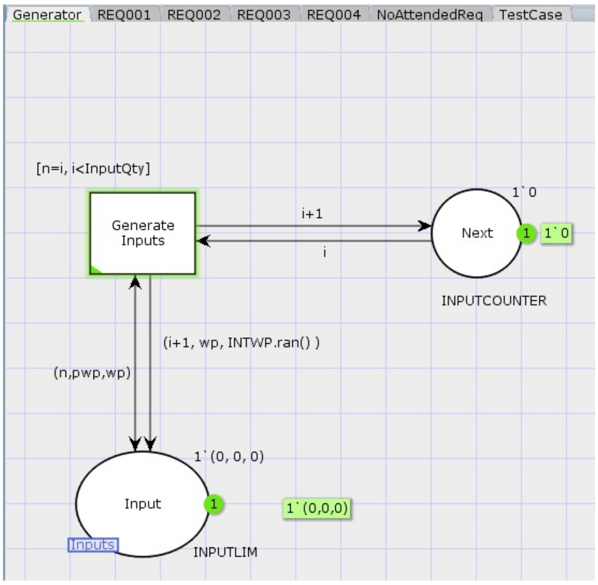


Fig. 4. Page created to generate the inputs randomly

of generated inputs reaches *InputQty*, the transition *Generate Inputs* becomes disabled. At this moment, all generated inputs (tokens) are stored in *Input*, and ready to be consumed by the other transitions.

Dealing with no system reaction A page (*NoAttendedReq*) similar to those that represent the requirements (DFRS functions) is created. Unlike those, although it has a transition (*noReq*) that consumes the inputs (tokens in the place *Input*) and increments the input ordering index, this transition does not change the value of the places that represent output variables. This transition is enabled when no other transition is enabled, and it represents the idea that when no system reaction is expected for a given input, the system does not change its outputs (their values remain the same). To ensure that *noReq* is only enabled when no other transition is enabled, it has a lower priority (*P_LOW*).

Ordering the generated test vectors While the place *Next Input* is used to define the order in which the inputs are processed, the place *Next TO* is used to order the obtained test vectors. When a transition related to a requirement (or the *noReq*) fires, it produces a token that represents a test vector: the received inputs, along with the expected results. As one can see in Figure 5, this token begins with an index z. This index, which is controlled by *Next TO*, establishes an order for the tokens stored in *Test Oracles*. Another element of this tuple is a label to track which requirement produced the output. This label allows us to relate generated test vectors with requirements and, thus, extract coverage information with respect to the system requirements.

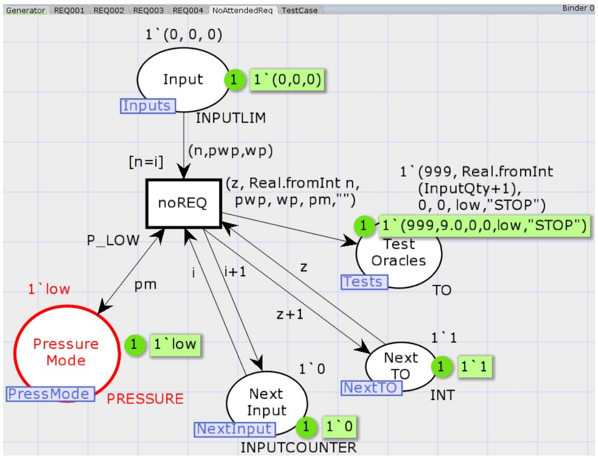
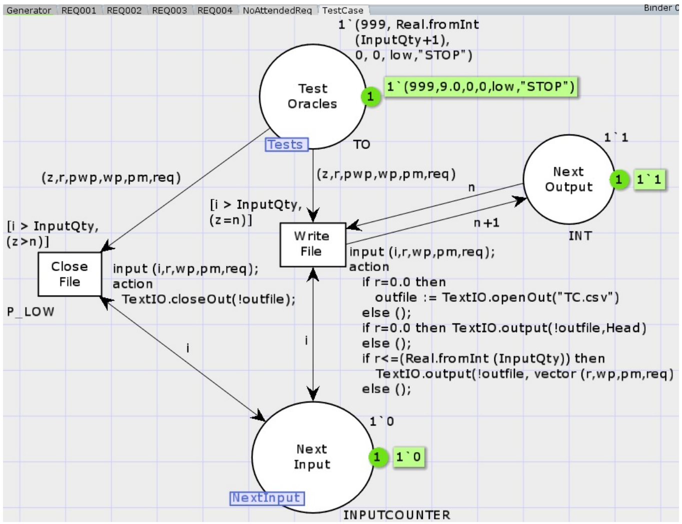


Fig. 5. Page created to deal with no system reaction

Creating test case file The page *TestCase* stores the steps (test vectors) of the generated test cases in a csv file. This page has two transitions to manipulate the csv files: *Write File* and *Close File*. The transition *Write File* is enabled when all inputs are processed, that is, when the index i (received from *Next Input*) is greater than the defined number of inputs of the test case. The created test vectors, which are stored in the place *Test Oracles* and ordered by the index z, are sent to this transition when it fires.

When fired, this transition executes the associated code segment (shown below) that creates a csv file. First, it creates a file (*TC.csv*), and adds to it a header – the name of the columns, which are named after the system input and output variables. Afterwards, it appends to this file a test vector every time the transition *Write File* is fired. The other transition (*Close File*) has lower priority and is enabled when all outputs are processed. When it fires, it closes the file previously created. These transitions are shown in Figure 6.

```
if r=0.0 then outfile := TextIO.openOut("TC.csv") else ();
if r=0.0 then TextIO.output(!outfile,Head) else ();
if r<=(Real.fromInt (InputQty)) then
   TextIO.output(!outfile, saida (r,wp,pm,req)) else ();
```



```
1`(999, Real.fromInt
   (InputQty+1),
   0, 0, low,"STOP")
```

```
[i > InputQty,
(z>n)]
```

```
[i > InputQty,
(z=n)]
```

```
input (i,r,wp,pm,req);
action
  TextIO.closeOut(!outfile);
```

```
input (i,r,wp,pm,req);
action
   if r=0.0 then
      outfile := TextIO.openOut("TC.csv")
   else ();
   if r=0.0 then TextIO.output(!outfile,Head)
   else ();
   if r<=(Real.fromInt (InputQty)) then
      TextIO.output(!outfile, vector (r,wp,pm,req)
   else ();
```

Fig. 6. Page created to save the test cases in a file

Repeated places Fusion sets are created for places that are repeated in more than one page. For instance, the place *Input* is contained in all pages, except in *TestCase*, so the fusion set *Inputs* is associated to all the places *Input* (see a thin rectangle at the bottom of this place in Figures 3, 4 and 5). The same occurs to the places *Test Oracles*, so the fusion set *Tests* is created. This place does not occur in the page *Generator*.

The simplicity behind the creation of these auxiliary components, which are in essence the same despite the system being modelled, shows how it is straightforward to extend the CPN that models a DFRS for the purpose of generating test cases.

4.2 Test Vectors Generation

Once generated the model, as previously explained in Section 3, it is possible to generate test cases automatically via CPN simulation: it randomly produces inputs, along with the expected outputs. The tool (CPN Tools) has a number of resources to run simulations; for instance, it can be performed with user intervention or automatically, it can also be repeated n times (simulation replications). Besides generating test cases, simulations can be used to validate the system requirements.

The proposed model is constructed so that the simulation takes place in three steps: generation of random inputs, processing the inputs by the transitions obtained from the system requirements (or by the transition *noREQ*), and recording the test case file. These steps are detailed below.

Entries generation In the first step, the inputs are randomly produced by using the *ran* function. As initial values are considered those defined in the DFRS, which are reflected as the initial marking of the place *Input*. The number of inputs is initially preset and stored in the constant *InputQty*. The entries are sorted with the support of place *Next*. Considering the NPP example, Figure 7 shows 7 entries randomly generated. Initially, the water pressure is 0. Later, it is 10, 4, 7, 2, 3, 1, 11. We note that the second element of the generated tokens refers to the previous value of the water pressure.

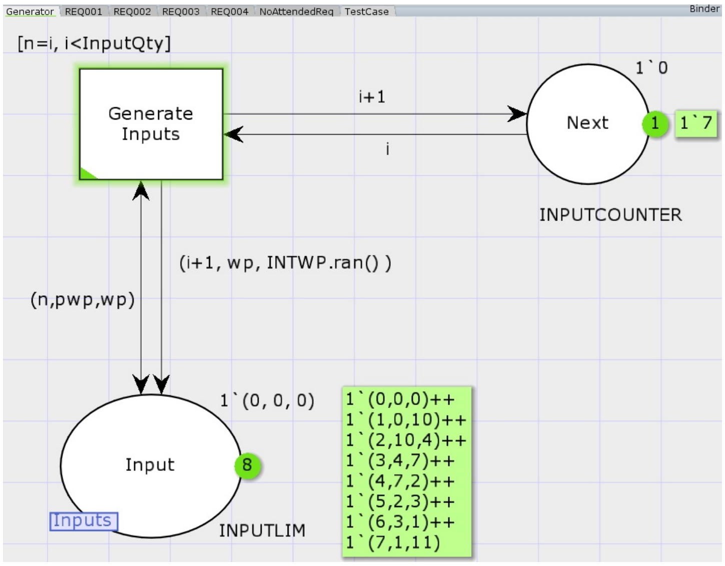


Fig. 7. Generating random entries

System reaction In the next step, for each input, the model verifies whether there is at least one enabled transition. If so, this transition is fired, and, as a result, it is obtained the expected system reaction. When firing a transition, it produces as output a tuple. This tuple comprises an ordering index, the previous and current values of the inputs and outputs, besides a label to track which requirements produced the output. The outputs are sorted with the support of the place *Next TO*. If for a given input there is no expected system reaction, the transition defined in the page *NoAttendedReq* is fired. It produces as output a new tuple keeping the output values the same. Considering the NPP example, Figure 8 shows three test vectors generated and stored in *Test Oracles* via CPN simulation.

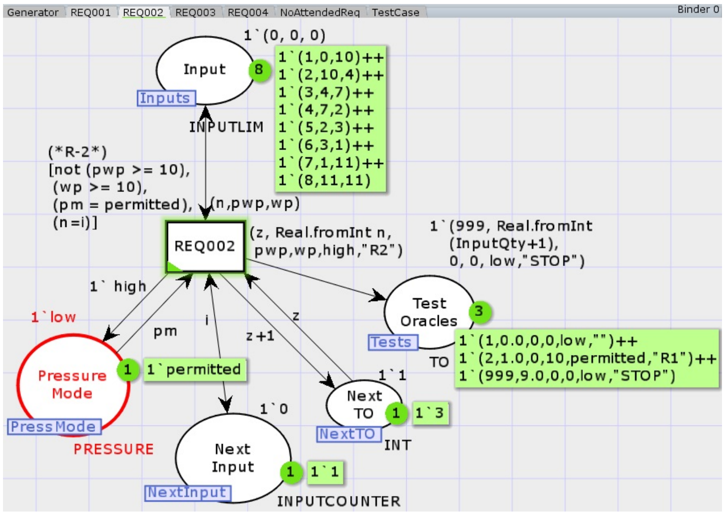


Fig. 8. Obtaining the expected system reaction

Test vectors Finally, in the last phase, the transition *Write File* records the test cases in a csv file. To accomplish this task, a *code segment* is used (shown in Section 4.1), which is executed when this transition fires. The place *Next Output* is used to keep the ordering of the outputs.

In the csv file, the generated test vectors are shown in a tabular form (see Table 1). The first column ($TIME$) is just the ordering index previously explained. The next two columns lists the system input, as well as the corresponding expected output. The last column provides traceability information between the expected output with the system requirements – *no req.* refers to a test vector that was generated by firing the transition *noReq*.

Initially, when the water pressures changes from 0 to 10 (time 0.0 and 1.0, respectively), first, the pressure mode changes to permitted (according to the requirement REQ001), then, immediately after, it changes to high (according to the requirement REQ002). Afterwards, there is no transition associated with a requirement that can fire, and thus, the transition *noReq* fires (repeating the last vector).

Therefore, as one can see, the test generation approach proposed here behaves according to the following cyclic pattern: first, it performs as many transitions related to requirements as possible; when all of these transitions are not enabled, it performs the *noReq* transition; finally, it receives new inputs.

The process for generating test cases via simulation of CPN does not involve complex and intensive computations. Particularly, the reached system states are dynamically computed considering the randomly generated inputs. For instance, this approach contrasts with strategies that need to explore the system complete

Table 1. Example of a test case for the simplified NPP

TIME	I: *the_water_pressure*	O: *the_pressure_mode*	REQ. Traceability
0.0	0	low	no req.
1.0	10	permitted	REQ001
1.0	10	high	REQ002
1.0	10	high	no req.
2.0	4	permitted	REQ004
2.0	4	low	REQ003
2.0	4	low	no req.
3.0	7	low	no req.
4.0	2	low	no req.
5.0	3	low	no req.
6.0	1	low	no req.
7.0	11	permitted	REQ001
7.0	11	high	REQ002
7.0	11	high	no req.
8.0	11	high	no req.

state space first and, thus, minimises state explosion problems. To generate more relevant test cases, the generation of inputs could be performed in a semi-random fashion in order to satisfy, for instance, user-defined coverage criteria. This improvement is a future work topic.

5 Conclusion

We presented in this work a variation of the NAT2TEST strategy that generates test cases from natural-language requirements via simulation of Colored Petri Nets (CPN). To accomplish this task, we define a systematic translation procedure that encodes the internal model of the NAT2TEST approach (*Data-Flow Reactive System* – DFRS) as a CPN. Afterwards, auxiliary components are added to the CPN to allow generation of test cases via simulation. Furthermore, this model is also suitable to analyses that might uncover problems in the original requirements, although this has not been the purpose of this paper.

There are several approaches to derive test cases from (semi-)formal models or directly from requirements expressed in some (controlled) natural language. Here we concentrate on approaches based on Petri nets.

A simple approach to generate test cases using CPN is proposed in [1]. The advantage of this approach is that the specification based on CPN can be validated by simulation (as in our case), but the state space analysis can lead to state explosion, unlike our approach, since the reached states are computed on demand. However, to perform analysis such as requirement completeness and consistency, it is likely that the complete state would need to be visited and, thus, our strategy would also fall into the state space problem.

An efficient approach for building a test suite using the PN-ioco conformance relation is proposed in [12]. Differently from our approach, it does not consider as input NL requirements. In [7], a method is proposed to convert UML 2.0 activity diagrams to a CPN model and apply the Random Walk Algorithm to create test sequences, differently from our strategy that is based on NL requirements. On the other hand, it considers coverage criteria when generating test cases.

In [16], two techniques are proposed for concurrent systems. One uses CPN-Tree and the other uses Colored Petri Net Graph (CP-graph). CPN-Graph is considered as finite state machines (FSM) and existing test case generation methods based on FSM are applied. In CPN-Tree method the reachability trees reduced by the equivalent marking technique are used to achieve the practical test suite length. Again, the techniques are dependent on the data. Therefore, the major drawback of the existing approaches are possible state space explosion (non-scalability), which is minimised in our approach since we generate test cases via simulation.

As future work, we intend to: (1) apply our approach to more complex examples, (2) compare the obtained results with the other variations of the NAT2TEST strategy, and (3) enhance our CPN model to consider temporal properties of the requirements, besides hybrid systems.

Acknowledgments. We thank Embraer for the partnership related to the NAT2TEST framework and, particularly, Braulio Horta and Ricardo Filho for their valuable contribution. This work was supported by the National Institute of Science and Technology for Software Engineering (INES[3]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.

References

1. Cai, L., Zhang, J., Liu, Z.: A CPN-based Software Testing Approach. JSW 6(3), 468–474 (2011)
2. Carvalho, G., Barros, F., Carvalho, A., Cavalcanti, A., Mota, A., Sampaio, A.: NAT2TEST Tool: from Natural Language Requirements to Test Cases based on CSP. In: International Conference on Software Engineering and Formal Methods. Springer International Publishing (2015)
3. Carvalho, G., Barros, F., Lapschies, F., Schulze, U., Peleska, J.: Model-Based Testing from Controlled Natural Language Requirements. In: Artho, C., lveczky, P.C. (eds.) Formal Techniques for Safety-Critical Systems, Communications in Computer and Information Science, vol. 419, pp. 19–35. Springer International Publishing (2014)
4. Carvalho, G., Carvalho, A., Rocha, E., Cavalcanti, A., Sampaio, A.: A Formal Model for Natural-Language Timed Requirements of Reactive Systems. In: Merz, S., Pang, J. (eds.) Formal Methods and Software Engineering, International Conference on Formal Engineering Methods ICFEM, Lecture Notes in Computer Science, vol. 8829, pp. 43–58. Springer International Publishing (2014)
5. Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Mota, A., Motta, L., Blackburn, M.: NAT2TEST$_{SCR}$: Test case generation from natural language requirements based on SCR specifications. Science of Computer Programming 95, Part 3(0), 275 – 297 (2014)

[3] www.ines.org.br

6. Carvalho, G., Sampaio, A., Mota, A.: A CSP Timed Input-Output Relation and a Strategy for Mechanised Conformance Verification. In: Formal Methods and Software Engineering, LNCS, vol. 8144, pp. 148–164. Springer Berlin Heidelberg (2013)
7. Farooq, U., Lam, C., Li, H.: Towards automated test sequence generation. In: Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on. pp. 441–450 (March 2008)
8. Jensen, K.: Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Springer-Verlag1 (1996)
9. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer Berlin Heidelberg (2009)
10. Jrgensen, J., Tjell, S., Fernandes, J.: Formal requirements modelling with executable use cases and coloured petri nets. Innovations in Systems and Software Engineering 5(1), 13–25 (2009)
11. Leonard, E., Heitmeyer, C.: Program synthesis from formal requirements specifications using apts. Higher-Order and Symbolic Computation 16(1-2), 63–92 (2003)
12. Liu, J., Ye, X., Li, J.: Colored Petri nets model based conformance test generation. In: Computers and Communications (ISCC), 2011 IEEE Symposium on. pp. 967–970 (June 2011)
13. Milner, R., Harper, R., Tofte, M.: The Definition of Standard ML. MIT Press (1990)
14. Myers, G., Sandler, C., Badgett, T.: The Art of Software Testing. John Wiley (2004)
15. Tjell, S.: Model-based Testing of a Reactive System with Coloured Petri Nets. In: Hochberger, C., Liskowsky, R. (eds.) Informatik. LNI, vol. 94, pp. 274–281. GI (2006)
16. Watanabe, H., Kudoh, T.: Test suite generation methods for concurrent systems based on coloured Petri nets. In: Software Engineering Conference, 1995. Proceedings., 1995 Asia Pacific. pp. 242–251 (Dec 1995)

Verifying Transformations of Java programs using Alloy

Tarciana Dias da Silva, Augusto Sampaio, and Alexandre Mota

Universidade Federal de Pernambuco - Centro de Informática
50740-560, Pernambuco, Brazil
{tds,acas,acm}@cin.ufpe.br

Abstract. In this paper we verify Java transformations by using a fourth–stage strategy. Initially we embed models in Alloy: a metamodel for a subset of the Java, a model for each program transformation being investigated, and another one for a program called *Validator* that exercises methods of each side of the transfomation. Secondly, we use the Alloy Analyzer to find valid instances, corresponding to pairs (left and right-hand sides of a program transformation) and instances of the *Validator*. If instances can be found, this means they describe well–formed programs as long as transformation conditions, structural and type constraints are formally stated in our models. Thirdly we developed a tool that translates the Alloy instances to Java; finally, these are executed and the results used to verify whether there are any dynamic semantic problems in the resulting programs.

Keywords: Java, Program Transformations, Alloy, Validation

1 Introduction

Program transformation (particularly, refactoring) is current practice in software development. Usually, such transformations are available in IDEs, like Eclipse or NetBeans. Unfortunately, in general neither the implementation nor a more abstract specification of the transformation itself are validated. This can cause behavioral changes as well as compilation errors after each transformation application, which is clearly unproductive.

Works such as [1, 2] validate tools that implement refactorings. In [1], the authors address the absence of precise refactoring specifications in modern available IDEs. They provide a high-level specification of common refactorings, but in terms of pseudocode, in order to facilitate implementations. The authors also compare their refactoring engine with the Eclipse one, using the Eclipse internal test suite. They also give a formal correctness proof for their refactorings.

The work in [2] presents a technique to test Java refactoring engines based on the generation of Java programs. The authors use Alloy for the generation of random instances (supported by a Java metamodel), and translate them to Java. In [2], after verifying that a generated instance is compilable, they apply a refactoring (available in some IDE) on such an instance and a new program is obtained. These programs (before and after a refactoring) are subjected to a test campaign, where behavioral changes, as well as compilation errors caused by the application of the refactoring, are evaluated. Unfortunately, both [1] and [2] have some drawbacks: when some test fails, it is not clear what the source of the problem is. It can be in the refactoring specification or in its implementation.

Complementarily to testing, formal languages provide a mathematically solid reasoning mechanism to establish the soundness of transformations. In [8], a set of algebraic laws is proposed for a subset of Java. Soundness is proved based on a formal semantics. In [9] a set of behavior-preserving transformation laws for a sequential object-oriented language is proposed with reference semantics (rCOS). The work described in [6] proposes laws in the Java context but neglects soundness proofs; the central barrier is the lack of a complete formal semantics for Java. The work in [10] enhances the one in [9] and proposes a set of algebraic laws for reasoning about object oriented programs with a reference semantics.

In the present work we propose a slightly different approach that combines (bounded) model finding and testing. We propose a strategy to validate Java program transformations grounded on a formal infrastructure built in Alloy [5]. Fig. 1 shows an overview of our strategy. In (1) we embed models in Alloy: a metamodel for a subset of the Java, a model for each transformation (law) being investigated and another one of a program generator that exercises methods of each side of the transformation; we call this a *Validator*. Apart from syntactic aspects, the embedded metamodel embodies a static semantics via constraints that ensures type correctness. Therefore, in our approach, the only potential issues caused by a transformation are behavioural ones. In (2), for a given transformation, we use the Alloy Analyzer to generate instances (representations) of the left–hand side (LS) and the right–hand side (RS) of the Alloy model of the transformation, as well as instances of the *Validator*. In (3), we use E-JDolly (an extension that we developed of the JDolly [2] tool) to translate these Alloy instances into concrete Java programs. If no valid instances are found, the transformation is unsatisfiable meaning that there is some predicate inconsistency, which denotes a specification error. In this direction, the Alloy Analyzer can detect syntax and static semantics problems. Finally, for each instance of the transformation model, we run the validator on the programs for LS and RS, and the results of the executions are compared; if they are not the same, then there is a behavioural non-conformance in the transformation, also meaning a transformation specification error. In this way, the *Validator* class can capture problems related to dynamic semantics. Preliminary results showed that transformation failures can be detected during the specification analysis, without the need to implement them in a source language or submitting them to a more elaborated test campaign as did in [1, 2]. Instead we use the Alloy Analyzer, with adequate models, and simple validating tests, in the *Validator* class.

In summary, the main contributions of this work are:

– a metamodel for a subset of Java in Alloy that is compliant to the Java Language Specification [3] and generates only compilable programs, differently, for instance, from [2];

- a model in Alloy representing each transformation. This model allows the generation of the left– and right–sides of a transformation (in the Alloy abstract syntax notation or Alloy instances format) and can be seen as a precise transformation specification;
- a formal transformation engine, E–JDolly, which translates the Alloy abstract syntax notation to the Java one;
- a Validation tool that checks (1) if all the constraints specified in the model are consistent, avoiding the generation of programs with structural or static semantics problems and (2) dynamic semantic problems, that cannot be identified by the Alloy Analyzer but are identified in our strategy by testing (through the *Validator* class).

Section 2 provides an overview of the algebraic style for presenting the transformation laws. A brief introduction to Alloy as well as a metamodel for a subset of Java in Alloy is presented in Section 3. Section 4 proposes a strategy for generating pairs of instances of programs corresponding to a given transformation as well as the generation of the *Validator* instances. Section 5 presents our extension to the JDolly framework, called *E–JDolly*. Section 6 describes how our tool can detect transformation specification errors which would cause compilation or behavioral problems. Section 7 addresses related work. Finally, Section 8 describes conclusions and topics for future investigation.

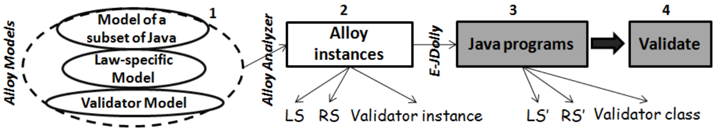


Fig. 1. Overview of the strategy.

2 Overview of the Algebraic Style

Typically, in the object–oriented paradigm, an algebraic law establishes the equivalence between two programs according to a program context represented by the entire set of class declarations (*cds*), a Main class (*Main*), and also considering that some provisos are respected. In the presentation style used in [8], also adopted here, conditions marked with (↔) must hold when performing the transformation in both directions; conditions marked with (→) must hold for the transformation from left to right, and those with (←) from right to left. In addition, $ads, cnds$, and mds represent the attribute, constructor, and method declarations, respectively; T represents an attribute type; and the symbol $\leq$ represents the subtype relation between classes.

Law 1 (Fig. 2) captures a transformation that moves an attribute to a superclass and also the inverse transformation (from right to left). This inverse transformation is used as one of the examples in this paper, where the attribute of class B is moved to class C. The proviso (←) of this law states that the attribute can be moved provided it does not already belong to the set of attributes of the class C (1). Besides, there must be no access to this attribute by any subclass of B, excluding the subclasses of C (2). The proviso (→) of this law is more simple: the attribute can be pulled up to class B provided there must be no declaration of this field in its subclasses. The constraints established by this algebraic law are reflected in the law–specific model (Fig. 1, (1)), detailed in Section 4.

Law 1 ⟨move attribute to superclass⟩

class B **extends** A {
ads
$cnds$
mds
}
class C **extends** B {
public T f; ads'
$cnds'$
mds'
}
	$=_{cds,Main}$	class B **extends** A {
		public T f; ads
		$cnds$
		mds
		}
		class C **extends** B {
		ads'
		$cnds'$
		mds'
		}

provided

(→) *The attribute name f is not declared by the subclasses of B in cds;*

(←) *(1) The attribute name f is not declared in ads'; (2) D.f, for any* $D \leq B$ *and* $D \nleq C$*, does not appear in cds, Main, cnds, cnds', mds, or mds'*

Fig. 2. Law 1 (move attribute to superclass and the inverse). Law 6 in [6]

Law 2 in Fig. 3 captures the moving of an original method to its superclass and the inverse transformation. This is similar to the *push down method* refactoring, mentioned in many works such as [1, 2]. The proviso (↔) states that the method can be pulled up or pushed down provided there is no access to *super* or *private* attributes in its body (1). In addition the method is not declared in any subclasses of B (2) and is not *private* (3). The proviso (→) requires there is no other method with the same signature and arguments in class B (right) (1) and the body of the method being pulled up do not contain any uncast occurrences of the keyword *this* or expressions in the form *((C)this)*.a, for any protected attribute a in the set of attributes of C class (2). Finally, proviso (←) requires there is no method declaration in C class (left) and (2) is similar to the condition (← (2)) of Law 1 (Fig. 2). As already mentioned, transformations in this style are the main input for our validation strategy.

3 Java metamodel embedding in Alloy

An Alloy specification can be represented by signatures, fields, constraints (facts, predicates or assertions) and functions. Each element declared as a *signature* represents a *type* and can also be associated to other elements by *fields* (or *relations*) along with their types. For instance, Fig. 4 shows a type *Class* (and other types it relates with) that owns the following fields: *id*, *extend*, *methods* and *fields* whose types, in turn, are, respectively: *ClassId*, *Class*, *Method*, *Field*. The keyword *one* in the declaration of the relation *id* indicates *multiplicity*—in this case it means that a type *Class* can only be associated with exactly one *ClassId*. Additionally, the relation *extend* associates the class declared in the signature with at most one element of type *Class*—this is ensured by the keyword *lone*. The relations *methods* and *fields* represent the set of elements of types *Method* and *Field*, respectively. Observe that *ClassId*, *MethodId* and *FieldId* are all subsignatures or

Law 2 ⟨move original method to superclass⟩

class B extends A {
  ads
  $cnds$
  mds
}
class C extends B {
  ads'
  $cnds'$
  $rt\ m(pds)\ \{mbody\}$
  mds'
}
	$=_{cds,Main}$	class B extends A {
		  ads
		  $cnds$
		  $rt\ m(pds)\ \{mbody\}$
		  mds
		}
		class C extends B {
		  ads'
		  $cnds'$
		  mds'
		}

provided

(↔) *(1)* **super** *and private attributes do not appear in mbody; (2)* $m(pds)$ *is not declared in any subclass of B in cds; (3)* $m(pds)$ *is not private.*

(→) *(1)* $m(pds)$ *is not declared in mds; (2) mbody does not contain uncast occurrences of* **this** *nor expressions in the form* $((C)$**this**$).a$ *for any protected attribute a in* ads'.

(←) *(1)* $m(pds)$ *is not declared in* mds'*; (2)* $D.m(e)$*, for any* $D \leq B$ *and* $D \nleq C$*, does not appear in* $cds, Main$*, mds or* mds'.

Fig. 3. Law 2 (move method to superclass and the inverse). Law 14 in [6]

extensions of type *Id*. Subsignatures in Alloy are subsets mutually disjoint of parent signatures. As can be seen, in our Java metamodel specification, the signatures and their respective relations are analogous to classes and associations in the UML class diagram (see Fig. 4—left).

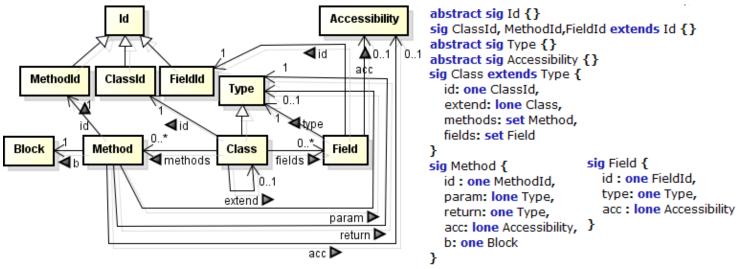
Code 1.1. Representation of the MethodInvocation signature

```
sig MethodInvocation extends StatementExpression {
    pExp: lone PrimaryExpression,
    methodInvoked: one Method,
    param: lone Type
}
```

Running the Alloy Analyzer on a specification creates an Alloy model (instances). Each Alloy instance is composed by the objects generated for each signature defined in the specification. The relationships among these objects rely on constraints, which are Java well-formedness rules in this paper. Fig. 4 (left) is a subset of our Java metamodel depicted in Fig. 5. It is used to show a connection between UML and Alloy. Code 1.1 shows *MethodInvocation* in Alloy. The *pExp* and *param* relations link a *MethodInvocation* with at most one *PrimaryExpression* and *Type*, respectively. In the last case, for simplification purposes. This constraint can be seen in the associations with *MethodInvocation* in Fig. 5.

3.1 Well-formedness rules

To ensure the well-formedness rules, some predicates and facts are specified in our Java metamodel like *noCalltoUndefinedField* (see Code 1.2). Such predicates avoid, for example, access to an undefined field. Predicate *noCalltoUndefinedField* states that in all cases where the access to the field is through a *PrimaryExpression* such as *newCreator* (*fa.pExp.cf* = *c*), then there are two possibilities: (1) the field belongs to the fields of the class *c* associated to the *newCreator* expression (*fa.fieldInvoked in c.fields*), and the field is *public* or it has no accessibility qualifier (*#(fa.fieldInvoked.acc)* = *0*)—for simplification purposes, we



```
abstract sig Id {}
sig ClassId, MethodId,FieldId extends Id {}
abstract sig Type {}
abstract sig Accessibility {}
sig Class extends Type {
   id: one ClassId,
   extend: lone Class,
   methods: set Method,
   fields: set Field
}
sig Method {
   id : one MethodId,
   param: lone Type,
   return: one Type,
   acc: lone Accessibility,
   b: one Block
}
sig Field {
   id : one FieldId,
   type: one Type,
   acc : lone Accessibility
}
```

Fig. 4. An UML class diagram and its representation in Alloy

assume that all classes are in the same package, there is no *Package* element in our model; or (2) the field belongs to the fields of some class in the parent level of class *c* (*(fa.fieldInvoked in (c.^extend).fields)*), and is not *private* nor with a default accessibility qualifier. On the other hand, when the access to the field is through a *PrimaryExpression* such as *this*, what changes is that the associated field must lie in the class itself; and when the access is through *super*, the field must lie in classes in the parent level of class *c* and must not be *private*.

Code 1.2. Predicate noCalltoUndefinedField

```
pred noCalltoUndefinedField[] {
  all c:Class, fa:FieldAccess | (fa.pExp.cf = c ) =>
      (fa.fieldInvoked in c.fields &&
         (fa.fieldInvoked.acc in public_|| #(fa.fieldInvoked.acc) = 0))
        ||
      ((fa.fieldInvoked in (c.^extend).fields) &&
        (fa.fieldInvoked.acc !in private_|| #(fa.fieldInvoked.acc) = 0))
  all c:Class, fa:FieldAccess | (fa.pExp in this_ ) =>
      (fa.fieldInvoked in c.fields )
  all c:Class, fa:FieldAccess | (fa.pExp in super_ ) =>
      (fa.fieldInvoked in c.fields && fa.fieldInvoked.acc !in private_)
}
```

Due to space limitation, it is not possible to explore all details of our Java metamodel specification but the complete code is publicly available.[1]. We define its elements as close as possible to the ones in the Java Language Specification [3]. Even considering a subset of the language and some simplifications, the well-formedness rules guarantee a 100% of compilable programs, different from the model presented in [2], which generates only a 68,8% of compilable programs.

4 Metamodel for Transformations and *Validator*

A second metamodel, for each algebraic law, is also specified in Alloy. This one uses the metamodel described in the previous section to represent the elements, such as classes and provisos, involved before and after the transformation described by the algebraic law (see Section 2). Hence, the Alloy Analyzer can simultaneously generate the Alloy instances for the programs on the left– and right–hand sides (as well as the instances representing the *Validator* for them) of each transformation.

[1] They can be downloaded from http://www.cin.ufpe.br/~tds/phd/JTransformations.

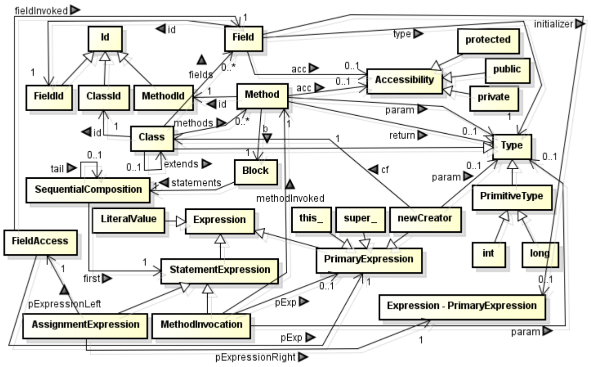


Fig. 5. Java metamodel embedded in Alloy

Fig. 6 illustrates a (simplified) set of related objects in an Alloy instance, generated by the law–specific Alloy model (Code 1.3). Recall that in Law 1 (Section 2), we have three classes A, B, and C. We can see object instances of different types such as type *Class* like *Class6* (which represents the A class in Law 1 (Fig. 2)), and subsignatures of type *Class* like *BRight*, *BLeft*, *CRight*, *CLeft* (see line 3, Code 1.3) and *Validator* (see line 3, Code 1.13); the ones in white dotted rectangles are subsignatures of type *id* like *ClassId_0* (of type *ClassId*) and *FieldId_0* (of type *FieldId*). The gray dashed ones are of type *Field* like *Field2* (the one that is being moved from right to left) and the ones in dotted dark gray are of type *Method*. Observe that the relations among all of these elements are represented by the arrows. For instance, the object instance *Class6* is in the *extend* relation of both *BLeft* and *BRight*. We opt to generate only one class (*Class6*), instead of two like *ALeft* and *ARight*, because there is no special rule concerned with the *A* classes and this minimizes the complexity of the Alloy model—by reducing the number of signatures and verifications.

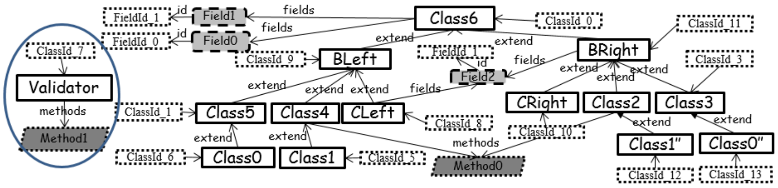


Fig. 6. Example of an Alloy instance generated, with its main elements

Code 1.3. Simplification of the Law 1 (Fig. 2) coding main part

```
1 module lawSpecificModel
2 open validatorModel
3 one sig BRight, CRight, BLeft, CLeft extends Class{}
4 pred law1RightToLeft[]{
5   twoClassesDeclarationInHierarchy[]
6   mirroringOfTwoClassesDeclarationsExceptForTheFieldMoved[]
7   law1[]
8 }
9 run law1RightToLeft for 10 but 15 Id, 15 Type, 15 Class
```

The Alloy predicate named *law1RightToLeft* (Code 1.3, lines 5 through 7) groups all the necessary predicates to be called in order to capture the transformation described in Law 1 (Fig. 2) (right to left). The first one, named *twoClassesDeclarationInHierarchy*, is shown in Code 1.4. It requires that there must be at least one class in the B's extend relation (lines 2 and 3), which is the A class—the left or the right–hand side (line 3). This is ensured in the *Class* signature (first metamodel, Section 3) because there the extend relation has at most one class (*extend: lone Class*). Besides, BRight is required to be in the CRight's extend relation (line 5) and BLeft in the CLeft's extend relation (line 6). In other words, CRight is the BRight's son and CLeft is the BLeft's son. In addition, these classes are required to have at least 2 sons (lines 15 and 16)—only to make the set of classes more interesting. The predicate called from lines 8 to 11, named *noMirrorClassExtendsTheOther*, is auxiliar and is used to avoid left classes to extend the right ones and vice-versa, because in fact these classes are the same but in different chronological order (before and after the transformation). For the same reason, the predicate *c1DoesNotContainc2Reference* is called (lines 13 and 14) to avoid left classes referencing right ones and vice-versa. The predicate in Code 1.4 is used in all laws that follow the structural pattern described in Law 1, with classes A, B and C in a hierarchy, which is also the case for Law 2.

Code 1.4. Predicate twoClassesDeclarationInHierarchy

```
1  pred twoClassesDeclarationInHierarchy[] {
2     some BRight.extend
3     some BLeft.extend
4     BRight.extend = BLeft.extend
5     BRight in CRight.extend
6     BLeft in CLeft.extend
7     all bl: BLeft, br: BRight, cl: CLeft, cr: CRight |
8        noMirrorClassExtendsTheOther[br, bl] &&
9        noMirrorClassExtendsTheOther[cr, cl] &&
10       noMirrorClassExtendsTheOther[br, cl] &&
11       noMirrorClassExtendsTheOther[cr, bl]
12    all c1:(*extend).BRight, c2: (*extend).BLeft
13       | c1DoesNotContainc2Reference [c1, c2] &&
14         c1DoesNotContainc2Reference [c2, c1]
15    #(extend.BRight) > 2
16    #(extend.BLeft) > 2
17 }
```

The next predicate in Code 1.3, named *mirroringOfTwoClassesDeclarationsExceptForTheFieldMoved* guarantees that left and right classes are the mirror from each other except for the field being moved by the transformation. This means that methods of BRight and BLeft, and those of CRight and CLeft, have a correspondence as well as their fields. This is also required for their extend relation as a simplification. Hence, there must exist a symmetry of the internal fields and methods of classes which are B's sons (except for the C's ones).

Finally, the last predicate in Code 1.3 is *law*1. It is shown in Code 1.5 and moves the field *f* from right to left through the predicate *movingASpecificFieldFromRightToLeft* and restricts access to this field through predicate *restrictFieldAccess*. The former is shown in Code 1.6 and ensures that the field *f* exists only in BRight and CLeft classes (lines 2 and 3), but not in the other classes (line 4). The latter is shown in Code 1.7 and is an embedding of the proviso (← (2)) of Law 1 (Fig. 2). It requires that classes that inherit from B (except for the C ones) do not have access to the specific field moved. Thus, the predicate *restrictFieldAccess* guarantees that there is no *FieldAccess* whose class in its *pExp* relation of *newCreator* type (see Fig. 5) is in the set of classes that are subtypes of BRight but not from CRight (predicate *classInBSubtypesButNotCSubtypes*), referenced in line 1 and detailed in Code 1.8.

Code 1.5. Predicate for Law 1 (Fig. 2)

```
1 pred law1 [] {
2     one f:Field | movingASpecificFieldFromRightToLeft[f] &&
3                   restrictFieldAccess[f]
4 }
```

Code 1.6. Predicate that moves the field from BRight to CLeft

```
1 pred movingASpecificFieldFromRightToLeft[f:Field] {
2   f in BRight.fields
3   f in CLeft.fields
4   all c:{Class-BRight-CLeft} | f !in c.fields
5   CRight.fields = CLeft.fields - f
6     BRight.fields - f = BLeft.fields
7 }
```

Code 1.7. Predicate that restricts the access to the field moved.

```
1 pred restrictFieldAccess[f:Field]{
2   no cfi:FieldAccess | classInBSubtypesButNotCSubtypes[cfi.pExp.cf]
3                        && cfi.fieldInvoked=f
4 }
```

Code 1.8. Predicate that states c is subtype of B (excluding C subtypes)

```
1 pred classInBSubtypesButNotCSubtypes[c:Class] {
2     c in (*extend.BRight - *extend.CRight) ||
3     c in (*extend.BLeft - *extend.CLeft)
4 }
```

Law 2 (Fig. 3) is defined in our Alloy model in almost the same way as Law 1 in Fig. 2, since they typically follow the same template. Code 1.9 shows the transformation from right to left, which is equivalent to the *push down method* refactoring. Predicate *mirroringOfTwoClassesDeclarationsExceptForTheMethod*, in line 3, is similar to the predicate *mirroringOfTwoClassesDeclarationsExceptForTheFieldMoved* explained earlier. So it is also the case of the predicates *movingASpecificMethodFromRightToLeft* and *restrictMethodAccess*, invoked in predicate *law2*, in line 3. The predicate *law2Proviso1* (Code 1.11) is the most significant difference between the specifications from Law 1 (Fig. 2) and Law 2 (Fig. 3). This predicate captures what is specified in proviso (↔) (1) in Law 2: no super or private attributes appear in the body of the method being moved. Since the only way for this fact to happen in our model is through an *AssignmentExpression*, then the predicate states that for all *AssignmentExpression* inside the body of the method being moved (*m.b.statements.((*tail).first))*)—line 2, Code 1.11–there is no *pExpressionLeft* of that assignment whose *pExp* relation (in case of a *MethodInvocation*) is in the *super* set of signature instances (line 2, Code 1.12) or no *pExpressionLeft* whose *fieldInvoked* (in case of a *FieldAccess*) has the *private* accessibility (line 3, Code 1.12). See also Fig. 5 to remember these relationships.

Code 1.9. Predicate that captures the transformation from right to left in Law 2.

```
pred law2RightToLeft[] {
  twoClassesDeclarationInHierarchy[]
  mirroringOfTwoClassesDeclarationsExceptForTheMethod[]
  law2[]
}
```

Code 1.10. Predicate *law*2.

```
pred law2[] {
  one m:Method | movingASpecificMethodFromRightToLeft[m] &&
                 restrictMethodAccess[m] && law2Proviso1[m]
}
```

Code 1.11. Predicate that captures the proviso (↔),Law 2.

```
1 pred law2Proviso1[m:Method] {
2   all ae:AssignmentExpression | (ae in m.b.statements.((*tail).first)) =>
3                        noSuperOrPrivateFieldAccess[ae.*pExpressionRight]
4 }
```

Code 1.12. Predicate that captures no access to super or private attributes.

```
1 pred noSuperOrPrivateFieldAccess[ae: AssignmentExpression]  {
2   ae.pExpressionLeft.pExp !in super_ ||
3   ae.pExpressionLeft.fieldInvoked.acc !in private_
4 }
```

The *Validator* model (simplified) is shown in Code 1.13. It states that the *Validator* class does not contain fields, sons or parents (line 5) and it only contains one method (line 6), that is public (line 7), unique for this class (line 8 and 9) and that cannot be invoked (lines 10 and 11) by other methods in other classes in the model. Since the method in the *Validator* instance will be translated to a *main* method, then for every *StatementExpression* inside it, its *PrimaryExpression* cannot be *this* or *super*. This is stated from lines 12 to 17.

Code 1.13. Simplification of the *Validator model.*

```
1  module validatorModel
2  open javametamodel
3  sig Validator extends Class{}
4  fact regardingValidatorClass{
5    #Validator.fields=0 && #(extend.Validator)=0 && #(Validator.extend)=0
6      && #(Validator.methods)=1
7    all m:Method | m in Validator.methods => m.acc in public
8    all m:Method | all c:{Class-Validator} | (m in c.methods) =>
9      m !in Validator.methods
10   one m:method | no mi:MethodInvocation |
11     m in Validator.methods && mi.methodInvoked = m
12   all ae:AssignmentExpression |
13     ae in Validator.methods.b.statements.((*tail).first) =>
14     noReferenceToThisOrSuperInAssignments[ae.*pExpressionRight]
15   all m:Method | all mi: MethodInvocation | m in Validator.methods &&
16      mi in (m.b).statements.((*tail).first) =>
17       mi.pExp !in this_ && mi.pExp !in super_
18 }
```

5 E–JDolly

E-JDolly is a Java program that translates the Alloy instances (see Section 4) into Java. E-JDolly redesigns JDolly [2]. Apart from the fact that our metamodel is significantly different from the one used by JDolly in [2], E–JDolly takes as input instances of a transformation specific model, as well as instances of the $Validator$. It then generates Java programs for the instances of left– and right–hand sides of the transformation, as well as for instances of the $Validator$; furthermore, all these are guaranteed to be compilable.

E–JDolly identifies related (left and right) Java classes from an Alloy transformation instance. For example, Fig. 6 shows an Alloy instance for our Alloy specification, as explained in Sections 3 and 4. In Fig. 6, Class4 and Class2, as well as Class5 and Class3, are related because they play corresponding roles on the left– and the right–hand sides. CLeft and CRight are the easiest to identify, as well as BRight and BLeft, because of their types. It is important to emphasize that object instances of type *Class* are always generated as pairs.

After identifying related classes in the Alloy instances, E–JDolly maps an Alloy abstract syntax tree into two Java abstract syntax trees (one for the left– and another for righ–hand side of the transformation). Later these two Java trees are stored in Java files, compiled and executed. Finally the *ids* of the corresponding classes are identified (as default, the ids of the left classes are used) because in the validation phase (see Section 6), they are executed by the *Validator* class in both program contexts (left and right). Fig. 7 shows the class diagram for Java classes generated by E–JDolly from the Alloy instance in Fig. 6.

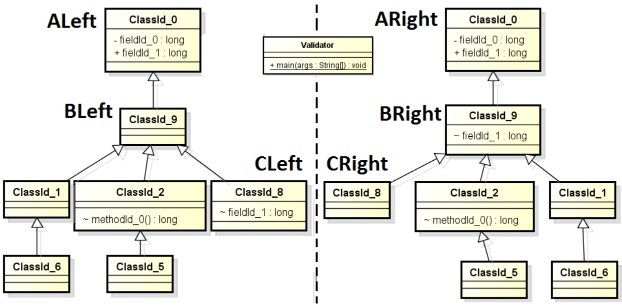


Fig. 7. Left and right classes representing the transformation in Law 1.

6 Validating the transformations

The specifications shown in Section 4 for Law 1 and Law 2 generate only well–formed pairs of programs (left– and right–hand sides of a transformation), according to the exaustive analysis provided by the Alloy Analyzer and within a given scope (line 9, Code 1.3). This guarantees no structural or static semantic problems in the pairs of programs as well as in the *Validator* class. This is confirmed by the *Validate* step (see (4) in Fig. 1) that compiles (in Java) each side of each transformation (after the translation of the *E–JDolly*) and no errors are found. In addition, the execution of the generated method in the *Validator* class in each side of each transformation also testifies the absence of dynamic execution problems.

In order to show that our strategy would detect structural and static semantic problems, suppose we change the body in the predicate in Code 1.8, which is referenced in Code 1.7, to the one presented in Code 1.14. In this new body, the B (or C) class is not included as its own subtype, only their children are, because there we are using transitive closure (ˆ) instead of a reflexive transitive closure (∗). This modification causes the Alloy Analyzer to consider the model as possibly inconsistent, and it does not generate any instances.

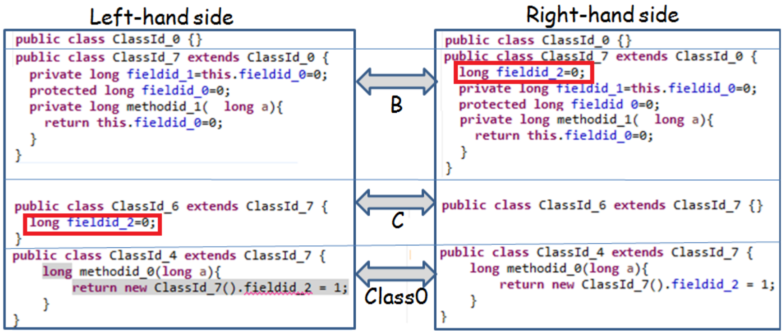
Code 1.14. Mutation of the predicate in Code 1.8

```
pred classInBSubtypesButNotCSubtypesWrong[c:Class] {
    c in (^extend.BRight - ^extend.CRight) ||
    c in (^extend.BLeft - ^extend.CLeft)
}
```

In order to better understand the error, first of all observe that the modification violates the proviso (←) of the Law 1 (Fig. 2). More specifically, *D*, mentioned in this proviso, assumes the value of a class of type *BRight* when it should not, because this class belongs to the set of the *B's* subtype—it is *B* itself. Mapping the situation to our model, a *FieldAccess*, using its *pExp* relation of type *newCreator* (a subtype of *PrimaryExpression*), whose *cf* relation, in turn, is of class *BRight* would be generated trying to access the specific field (relation *fieldInvoked* of *FieldAccess* signature) being moved by the transformation. However, we have predicates that guarantee the symmetry among left and right classes, as mentioned in Section 4, which implies in having the same *FieldAcess* element signature, but with *cf* relation of class *BLeft* instead (in left–hand side classes). This evaluation together with the predicate *noCallToUndefinedField* (Code 1.2) is contraditory since there is no field in the class *BLeft*, only in *BRight*. Thus, programs such as the ones presented in Fig. 8 are not generated. This program shows the class *ClassId_4* (*left*) trying to access the field (*fieldid_2*) (It is no longer available because it was moved from the right to the left–hand side). We have explored several variations of the transformation that generate syntactic errors, and in all cases our Alloy model has not generated any instances, which gives some evidence that the metamodel embedded in Alloy indeed captures the syntax and static semantics of the considered Java subset.

With regard to dynamic semantic problems, suppose we change the specification of Law 2, shown in Section 4, to apply a typical rule used in the *push down method* refactoring: when, inside the body of a method to be pushed down, there is an access to another method in the same class (through the keyword *this*), then we change the access from *this* to *super*. The intention is to give some flexibility to the application of the refactoring, instead of rejecting its application or avoiding compilation errors if we consider the opposite direction: the *pull up* method or the transformation in Law 2 from left to right. Then some adjustments in the input program are performed.

As our model in Alloy is faithful to the algebraic law, a pair of programs to represent this scenario would never be generated because this flexibility is not contemplated. However, in order to show that a behavioral problem would be found by our solution (when this flexibility is applied to the specification in our Alloy model), we include some mutation as seen in Code 1.15.



Left-hand side:

```
public class ClassId_0 {}
public class ClassId_7 extends ClassId_0 {
  private long fieldid_1=this.fieldid_0=0;
  protected long fieldid_0=0;
  private long methodid_1(  long a){
    return this.fieldid_0=0;
  }
}

public class ClassId_6 extends ClassId_7 {
  long fieldid_2=0;
}
public class ClassId_4 extends ClassId_7 {
    long methodid_0(long a){
        return new ClassId_7().fieldid_2 = 1;
    }
}
```

Right-hand side:

```
public class ClassId_0 {}
public class ClassId_7 extends ClassId_0 {
  long fieldid_2=0;
  private long fieldid_1=this.fieldid_0=0;
  protected long fieldid_0=0;
  private long methodid_1(  long a){
    return this.fieldid_0=0;
  }
}

public class ClassId_6 extends ClassId_7 {}

public class ClassId_4 extends ClassId_7 {
    long methodid_0(long a){
        return new ClassId_7().fieldid_2 = 1;
    }
}
```

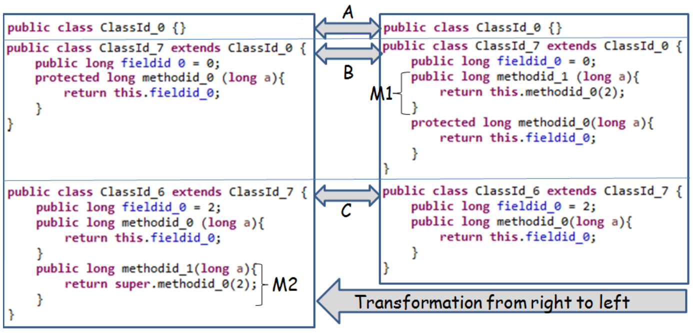
Fig. 8. Classes showing anomalies in the program transformation.

Observe that, due to the change from *this* to *super* in some *MethodInvocation* calls inside the body of the method being pushed down, the bodies (right and left) of the method being moved are different; so we have to create two distinct method signatures in Alloy (line 1, Code 1.15)— method *M1* on the right–hand side and *M2* on the left. Note that the main difference of this specification to the one discussed in Section 4 is in the predicate *mutation* (lines 21 to 27, Code 1.15), where we guarantee for each *MethodInvocation* (in the *M1* body) whose *pExp* relation (type *PrimaryExpression*) is in the *this* set of signature instances (line 23) that there is another one (inside the *M2* body) whose same relation (*pExp*) is in the *super* set of signature instances (line 24).

With this new specification for Law 2 (from right to left), programs such as the one in Fig. 9 are generated, with behavioral problems since, before the transformation, when for instance we invoke the method *methodid_1* from *ClassId_6* (*new ClassId_6().methodid_1()*), the result was 2 and, after the transformation, is 0. This occurs because, before, as *methodid_1* is invoked on an instance of *ClassId_6* (*new ClassId_6().methodid_1()*), which in turn invokes *methodid_0()*, using the qualifier *this*, the keyword *this* refers to the implementation of the method *methodid_0* in *ClassId_6*, which yields 2. On the other hand, after the transformation, *methodid_1()* invokes *methodid_0* of the super class of the *ClassId_6* class, which yields 0. Observe that the test itself (*new ClassId_6().methodid_1()*) does not need to be sofisticated to catch behavioral problems but instead the programs where these tests would be applied.

Code 1.15. Mutation of the Law2 specification

```
1 one sig M1, M2 extends Method {}
2 pred law2RightToLeftIncorrect [] {
3   twoClassesDeclarationInHierarchy []
4   mirroringOfTwoClassesDeclarationsExceptForTheMethod []
5   law2Mutation []
6 }
7 pred law2Mutation [] {
8       movingASpecificMethodFromRightToLeftMutation [] &&
9       restrictMethodAccessRight [M1] && restrictMethodAccessLeft [M2] &&
    law14Proviso1[M1] && law14Proviso1[M2]
}
pred movingASpecificMethodFromRightToLeftMutation[] {
  M1 in BRight.methods
  M2 in CLeft.methods
  all c:{Class-BRight} | M1 !in c.methods
  all d:{Class-CLeft} | M2 !in d.methods
  CRight.methods = CLeft.methods - M2
    BRight.methods - M1 = BLeft.methods
  mutation[]
}
pred mutation [] {
    all mi: MethodInvocation | some mi2:MethodInvocation |
        (mi in M1.b.statements.(*tail).first && mi.pExp in this_) =>
        (mi2 in M2.b.statements.(*tail).first && mi2.pExp in super_ &&
        equalsMethodInvocation[mi, mi2] &&
        equalsMethod[M1, M2, mi, mi2])
}
```



```
public class ClassId_0 {}
public class ClassId_7 extends ClassId_0 {
    public long fieldid 0 = 0;
    protected long methodid_0 (long a){
        return this.fieldid_0;
    }
}
public class ClassId_6 extends ClassId_7 {
    public long fieldid_0 = 2;
    public long methodid_0 (long a){
        return this.fieldid_0;
    }
    public long methodid_1(long a){
        return super.methodid_0(2);
    }
}
```

```
public class ClassId_0 {}
public class ClassId_7 extends ClassId_0 {
    public long fieldid_0 = 0;
    public long methodid_1 (long a){
        return this.methodid_0(2);
    }
    protected long methodid_0(long a){
        return this.fieldid_0;
    }
}
public class ClassId_6 extends ClassId_7 {
    public long fieldid_0 = 2;
    public long methodid_0(long a){
        return this.fieldid_0;
    }
}
```

Fig. 9. Classes generated according to incorrect specification shown in Code 1.15.

7 Related Work

Many authors refer to preconditions incompleteness as the main source of program misbehavior after refactoring application. In [12] a library containing preconditions and ways to check them is available in different languages. A type constraint–based approach is proposed in [13] to check preconditions of refactorings. We complement both works as our strategy allows one to specify a transformation (where preconditions are a mandatory part–see Section 2) and validate formally.

Expressing transformations correctly for complex languages like Java is difficult. Algebraic laws help by focusing on compositionality. A common feature of all the works in this direction, described in Section 1, is that they are based on extremely simplified languages, when compared to languages like Java. The advantage is that these languages have a formal semantics and allow one to prove soundness of the transformations.

Apart from what was mentioned about the work in [2], this work uses the Saferefactor tool [7] to assert whether there are any behavioral discrepancies among the left– and right–hand side programs (this last one generated by some

IDE plugin implementation). When a test fails, its origin can be the specification or the implementation engine. As aforementioned, the work in [1] provides a high-level specification of common refactorings. Their specification are based on the concepts of dependency preservation, language extensions, and microrefactorings. Their refactoring engine is implemented as an extension to the JastAddJ Java compiler [14]. In addition, the engine is verified using both correctness proofs and three test suites: their own, one for Eclipse and one for IntelliJ (publicly available). Despite this, the work [2] found bugs in the engine presented by [1]. Nevertheless, again, it is not known if the error comes from a specification or implementation flaw. In our case the flaw is always in the refatoring specification.

8 Conclusions and Future Work

This work presents a strategy for validating Java program transformations and algebraic laws relying on formal verification techniques (particularly, Alloy and the Alloy Analyzer). The strategy consists in building a metamodel for a subset of Java along with a model for each transformation (law) being investigated and for a program generator (the Validator) that exercises methods of each side of the transformation, through a random sequence of statements generated inside its *main* method. The Alloy Analyzer produces, from these models, Alloy instances that represent a set of classes in Java involved in a transformation so that it is possible to identify the ones *before* and *after* the transformation. *E–JDolly* translates the Alloy instances into Java programs. Afterwards, the Validator class is executed and its *main* method applied in the context of the classes generated for the left– and the right–hand sides of a transformation. This enables detecting behavioral errors; compilation errors are inherently avoided by our metamodel that considers both syntactic and static semantic analysis.

The Alloy Analyzer acts, with an adequate model, as a powerful test generator since it generates, according to the transformation scenario (modeled in law–specific model, Section 4), different possibilities of classes, their relationships and properties such as methods and attributes, elements inside method bodies as well as the other properties included in our Java metamodel (Section 3). Thus, structural problems are avoided through the own specification whilst with simple validating tests, as done with the class *Validator*, Section 6, behavioral problems can also be detected, similar to the ones shown in [1, 2]. Once the specification is validated, then a developer can implement the transformations with some confidence. In addition, as a future work, our refactoring Alloy implementation can be easily coupled in an IDE for implementing transformations in specific programs. It only requires an additional translator: from Java to Alloy, in such a way that this process would be transparent to the IDE user.

Our experiments so far helped to validate our own metamodel and the algebraic laws proposed in [8], and adapted for Java in [6]. Despite the results already achieved, there are some interesting directions for extending the proposed strategy. As a next step we aim at enhancing our strategy by adding steps to consider LS' (Fig. 1) as an input to refactoring engines such as Eclipse. By applying the engine refactoring implementation, we get a new right–side program (RS") in Java. We then translate this Java code back to Alloy and compare it with LS' within the Alloy Analyzer. This allows a way of checking the soundness of refactoring engines in a more rigorous way than testing. Besides, we intend to design a controlled experiment to further compare our strategy with the ones proposed in [1, 2].

Acknowledgments. This work was supported by 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.

References

1. Schäfer, M. Specification, Implementation and Verification of Refactorings, Phd Thesis, 2010.
2. Soares, G., Gheyi, R., Massoni, T: Automated Behavioral Testing of Refactoring Engines. In: IEEE TSE Transactions on Software Engineering, vol. 39, number 2, pp 147–162. (2013)
3. Java Language Specification, http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf
4. Jackson, D., Schechter, I., and Shlyahter, I.: Alcoa: the Alloy constraint analyzer. In: 22nd ICSE International Conference on Software Engineering, pp 730–733. ACM Press, New York (2000)
5. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press. (2006)
6. Duarte, R., Mota, A., Sampaio, A.: Introducing concurrency in sequential Java via laws. In: Information Processing Letters. vol. 111, issue 3, pp 129–134, elsevier. (2011)
7. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making program refactoring safer. In: IEEE Software, vol. 27, pp 52–57. (2010)
8. Borba, P., Sampaio, A., Cavalcanti, A., and Cornelio, M.: Algebraic reasoning for object-oriented programming. In: Science of Computer Programming, vol. 52, pp 53–100 (2004).
9. Silva, L., Sampaio, A., and Liu, Z.: Laws of object-orientation with reference semantics. In: 6th IEEE SEFM International Conference on Software Engineering and Formal Methods, pp 217–226, Washington (2008).
10. Palma, G. Algebraic Laws for Object Oriented Programming With References, Phd Thesis, 2015.
11. Naumann, D., Sampaio, A., Silva, L.: Refactoring and representation independence for class hierarchies. In: Theoretical Computer Science, vol. 433, pp 60–97 (2012).
12. Overbey, J.L., Johnson, R.E.: Differential precondition checking: A lightweight, reusable analysis for refactoring tools. In: 26th IEEE/ACM ASE International Conference on Automated Software Engineering, pp 303–312, New York (2011).
13. Tip, F., Kiezun, A. and Baumer, D.: Refactoring for generalization using type constraints. In: 18th ACM SIGPLAN OOPSLA Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp 13–26, New York (2003).
14. Ekman, T. and Hedin, G.: The JastAdd Extensible Java Compiler. In Object–Oriented Programming, Systems and Languages (OOPSLA), pages 1–18. ACM Press (2007)

[2] www.ines.org.br

Instantiation Reduction in Iterative Parameterised Three-Valued Model Checking

Nils Timm and Stefan Gruner

Department of Computer Science, University of Pretoria, South Africa
{ntimm,sgruner}@cs.up.ac.za

Abstract. We introduce an enhanced approach to parameterised three-valued model checking (PMC) based on *iterative* parameterisation. The model is parameterised until it is precise enough for a definite verification result. Results from past iterations are *reused* to reduce the number of parameter instances in future iterations. Our approach is based on a SAT encoding. In the initial iteration we construct an over-approximation of all possible instances in later iterations. For this over-approximation we compute the set of all satisfying interpretations. All subsequent iterations are then accomplished by *validating* whether for each instance one of the precomputed interpretations is satisfying as well, which is less costly than solving each SAT instance from scratch. Our iterative parameterisation approach leads to a substantial speed-up of PMC.

1 Introduction

Three-valued abstraction [11] is an established technique in software verification. It proceeds by generating a state space model of the system to be analysed over the values *true*, *false* and *unknown*, where the latter value represents the loss of information due to abstraction. The evaluation of temporal logic properties on such models is *three-valued model checking* (3MC) [3]. In case of an *unknown* result, the abstraction is typically refined by adding more predicates over the systems variables until a level of abstraction is reached where the property can be definitely proved or refuted. With each additional predicate the state space grows *exponentially.* Thus, such a classical refinement does not guarantee that eventually a model can be constructed that is both precise enough for a definite outcome and small enough to be manageable with the available resources.

At SBMF '14 we introduced *parameterised three-valued model checking* (PMC) [12] as an extension of 3MC. Predicates and transitions in PMC models can be associated with the values *true, false, unknown*, or with expressions over *Boolean parameters.* Parameterisation is an alternative way to state that the value of certain predicates or transitions is not known and that the checked property has to yield the same result under each possible parameter instantiation. PMC is thus conducted via evaluating a property on *all* instantiations, whose number is *exponential* in the number of parameters. Parameterisation particularly allows to establish *logical connections* between unknowns in the model: It enables to represent facts like 'a certain pair of transitions has unknown but *complementary* truth values'. Such facts can be automatically derived from the system to be verified, and covering these facts in a model can significantly enhance the precision in verification. In many cases it is possible to replace a large number of classical refinement steps by a single parameterisation step in order to obtain the necessary precision for a definite result. In [13] we introduced a propositional logic encoding of PMC problems that allows to perform PMC via SAT solving. The solving performance of our SAT-based approach profits from conflict clause sharing between the SAT instances. However, the number of instances associated with an encoding is still *exponential* in the number of parameters.

Here we introduce an enhanced approach to PMC that substantially overcomes the exponential overhead of parameterisation that existed in [12, 13]. Parameterisation is typically applied *iteratively* until the model is precise enough for a definite result. We show that in an iterative approach partial results from past iterations can be used to narrow the number of relevant instantiations in future iterations. Technically, a parameterisation step splits each instantiation of the current iteration into two instantiations of the subsequent iteration. This allows us establish a successor relation between the instantiations of different parameter iterations. We prove that if a property holds for a certain instantiation of the current iteration then it also holds for all its successor instantiations in future iterations. Hence, in each iteration only those instantiations have to be considered for which no predecessor exists that fulfils the property. Such an *instantiation reduction* works for *disproving* temporal logic properties as well.

Moreover, our iterative approach allows to significantly reduce the computational costs for checking *individual* instantiations. The initially non-parameterised model can be straightforwardly transferred into an over-approximation (wrt. the validity of temporal logic properties) of all parameter instantiations in later iterations. We exploit this fact as follows: In the first iteration we compute the set of all paths witnessing the property of interest in the over-approximation. In each subsequent iteration, instead of conducting model checking from scratch for each instantiation, we only have to track the paths from the over-approximating set in the instantiations in order to determine whether any of them witnesses the property here too. The savings of this approach become evident in our SAT-based PMC scenario: The computation of all witness paths in the over-approximation reduces to the computation of all satisfying interpretations of the initial encoding, which can be efficiently performed via AllSAT techniques [15, 8, 14]. Now all subsequent iterations can be accomplished by just *validating* whether for each SAT instantiation one of the precomputed interpretations is satisfying as well.

Our new concepts *instantiation reduction* and *interpretation validation* thus enhance PMC on the one hand by reducing the *number* of instantiations that have to be processed per iteration, and on the other hand by reducing the *effort* for processing individual instantiations. This enables us to profit from the extra precision of parameterisation without suffering from the previous drawbacks. We implemented a bounded model checker for PMC problems that employs iterative parameterisation and applies our optimisations. Preliminary experiments show that our new approach leads to a substantial speed-up of PMC.

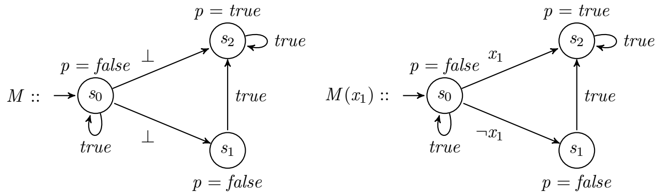
2 Background: Parameterised 3-Valued Model Checking

We briefly review *pure* three-valued model checking (3MC) [3] and *parameterised* three-valued model checking (PMC) [12]. A more extensive introduction can be found in [12, 13]. The key feature of 3MC is a third truth value $\perp$ (i.e. *unknown*) for transitions and labels, which is used to model uncertainty. PMC additionally allows *Boolean parameter expressions*, which enable to establish logical connections between unknown parts. As models we use Kripke structures:

Definition 1 (Parameterised Three-Valued Kripke Structure). *A* parameterised three-valued Kripke structure *over a set of atomic predicates AP and an ordered set of Boolean parameters $X_n = (x_1, \ldots, x_n)$ is a parameterised tuple $M(X_n) = (S, s_0, R(X_n), L(X_n))$ where*

- *S is a finite set of states and $s_0 \in S$ is the initial state,*
- *$R(X_n) : S \times S \to \{true, \perp, false\} \cup BE(X_n)$ is a transition function with $\forall\, s \in S : \exists\, s' \in S : R(X_n)(s, s') \in \{true, \perp\} \cup BE(X_n)$ where $BE(X_n)$ denotes the set of Boolean expressions over X_n,*
- *$L(X_n) : S \times AP \to \{true, \perp, false\} \cup BE(X_n)$ a label function that associates a truth value or parameter expression with each predicate in each state.*

An *instantiation* of a parameterised three-valued Kripke structure $M(X_n)$ is a *pure* three-valued Kripke structure $M(B_n)$ where $B_n \in \{true, false\}^n$. A structure is also *pure* if $X_n = \varnothing$. An example for a pure three-valued Kripke structure M and a parameterised Kripke structure $M(x_1)$ is depicted below.



A parameterised three-valued Kripke structure can be obtained by parameterising a pure three-valued Kripke structure according to the rules from [12]. This enhances the precision but also leads to an exponential increase of the number of parameter instantiations that have to be considered. For instance, if the transitions (s_0, s_1) and (s_0, s_2) of our example structure M correspond to a complementary branch (e.g. *if-then-else*) in the modelled system, then $M(x_1)$ is a sound parameterisation of M.

In the following we first introduce 3MC and then generalise it to PMC. A path π of a structure M is a sequence $s_0 s_1 s_2 \ldots$ with $R(s_i, s_{i+1}) \in \{true, \perp\}$. $\pi(i)$ denotes the i-th state of π, whereas π^i denotes the i-th suffix $\pi(i)\pi(i+1)\ldots$ of π. By Π_M we denote the set of all paths of M starting in the initial state. We use the temporal logic LTL for specifying properties with regard to paths.

Definition 2 (Syntax of LTL). *Let AP be a set of atomic predicates and $p \in AP$. The* syntax of LTL formulae *ψ is given by*

$$\psi \ ::= \ p \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathbf{X}\psi \mid \mathbf{G}\psi \mid \mathbf{F}\psi \mid \psi\mathbf{U}\psi.$$

In 3MC we operate under the three-valued Kleene logic $\mathcal{L}_3$ [7]. Based on $\mathcal{L}_3$, LTL formulae can be evaluated on paths Kripke structures:

Definition 3 (Three-Valued Evaluation of LTL). *Let $M = (S, s_0, R, L)$ over AP be a pure three-valued Kripke structure. Then the* evaluation *of an LTL formula ψ on a path $\pi \in \Pi_M$, written $[\pi \models \psi]$, is defined as follows*

$$\begin{array}{lll}
[\pi \models p] & := & L(\pi(0), p) \\
[\pi \models \neg\psi] & := & \neg\, [\pi \models \psi] \\
[\pi \models \psi \vee \psi'] & := & [\pi \models \psi] \;\vee\; [\pi \models \psi'] \\
[\pi \models \mathbf{G}\psi] & := & \bigwedge_{i\in\mathbb{N}}(R(\pi(i), \pi(i+1)) \wedge \left[\pi^i \models \psi\right]) \\
[\pi \models \mathbf{F}\psi] & := & \bigvee_{i\in\mathbb{N}}(\left[\pi^i \models \psi\right] \wedge \bigwedge_{j=0}^{i-1} R(\pi(j), \pi(j+1)))
\end{array}$$

Complete LTL definitions can be found in [12]. 3MC [3] is now defined as follows:

Definition 4 (Three-Valued LTL Model Checking). *Let $M = (S, s_0, R, L)$ over AP be a three-valued Kripke structure. Moreover, let ψ be an LTL formula over AP. The* universal value *of ψ in M is $[M \models_U \psi] \;:=\; \bigwedge_{\pi\in\Pi_M} [\pi \models \psi]$. The* existential value *of ψ in M is $[M \models_E \psi] \;:=\; \bigvee_{\pi\in\Pi_M} [\pi \models \psi]$.*

Universal model checking can always be transferred into existential model checking based on the equation $[M \models_U \psi] = \neg\,[M \models_E \neg\psi]$. In 3MC there exist three possible outcomes: *true*, *false* and $\perp$. For our example structure M we get $[M \models_U \mathbf{G}\neg p] = \neg[M \models_E \mathbf{F}p] = \perp$. Hence, M is not precise enough for a definite result. From now on, we only consider the existential case, since it is the basis for SAT-based bounded model checking [2] which we later apply.

3MC can always be reduced to two instances of two-valued model checking (2MC) if the LTL formula is restricted to LTL$^+$, i.e. negation-free LTL. LTL$^+$ formulae are evaluated on *complement-closed* structures. In these structures each $p \in AP$ has a complementary $\bar{p} \in AP$ such that $L(s,p) = \neg L(s,\bar{p})$. Every structure can be extended to a complement-closed one, without increasing the number of states. For the evaluation on complement-closed structures, each LTL formula can be transferred into an equivalent LTL$^+$ formula. Thus, the restriction to LTL$^+$ does not limit the expressiveness. The reduction of 3MC to 2MC is based on *completions* of complement-closed structures.

Definition 5 (Completion). *Let $M = (S, s_0, R, L)$ over AP be a three-valued Kripke structure. Then $M^p = (S, s_0, R^p, L^p)$ is the* pessimistic completion *and $M^o = (S, s_0, R^o, L^o)$ is the* optimistic completion *with $\forall\, s, s' \in S$ and $\forall\, p \in AP$:*

$$L^p(s,p) := \begin{cases} \mathit{false} & \mathit{if}\ L(s,p) = \perp \\ L(s,p) & \mathit{else} \end{cases} \qquad R^p(s,s') := \begin{cases} \mathit{false} & \mathit{if}\ R(s,s') = \perp \\ R(s,s') & \mathit{else} \end{cases}$$

$$L^o(s,p) := \begin{cases} \mathit{true} & \mathit{if}\ L(s,p) = \perp \\ L(s,p) & \mathit{else} \end{cases} \qquad R^o(s,s') := \begin{cases} \mathit{true} & \mathit{if}\ R(s,s') = \perp \\ R(s,s') & \mathit{else} \end{cases}$$

The following lemma from [3] establishes the reduction from 3MC to 2MC:

Lemma 1. *Let $M = (S, s_0, R, L)$ be a complement-closed three-valued Kripke structure and ψ an LTL$^+$ formula. Then the following holds:*

$$[M \models_E \psi] = \begin{cases} \textit{true} & \textit{iff } [M^p \models_E \psi] = \textit{true} \\ \textit{false} & \textit{iff } [M^o \models_E \psi] = \textit{false} \\ \perp & \textit{else} \end{cases}$$

Hence, if a formula holds for the pessimistic completion then it also holds for the three-valued Kripke structure. The same applies to *false* results obtained for the optimistic completion. All definitions wrt. pure 3MC can be straightforwardly generalised to PMC [12], since PMC reduces to multiple instances of 3MC.

Definition 6 (Parameterised Three-Valued LTL$^+$ Model Checking). *Let $M(X_n) = (S, s_0, R(X_n), L(X_n))$ be a parameterised three-valued Kripke structure over AP and $X = (x_1, \ldots, x_n)$. Moreover, let ψ be an LTL$^+$ formula over AP. The existential value of ψ in $M(X_n)$, written $[M(X_n) \models_E \psi]$, is defined as*

$$[M(X_n) \models_E \psi] := \begin{cases} \textit{true} & \textit{if } \forall\, B_n \in \{t, f\}^n \; ([M(B_n)^p \models_E \psi] = \textit{true}) \\ \textit{false} & \textit{if } \forall\, B_n \in \{t, f\}^n \; ([M(B_n)^o \models_E \psi] = \textit{false}) \\ \perp & \textit{else} \end{cases}$$

We call $M(B_n)^p$ a *pessimistic instantiation* where the parameters are instantiated wrt. B_n and the $\perp$'s are set to *false*. In an *optimistic instantiation* $M(B_n)^o$ the $\perp$'s are set to *true* instead. If checking a property yields *true* for all pessimistic instantiations, this result is transferred to the parameterised structure. The same holds for *false* results for all optimistic instantiations. In all other cases PMC returns $\perp$. For our example $M(x_1)$, we get $[M(x_1) \models_E \mathbf{F}p] = \textit{true}$ since $\mathbf{F}p$ holds existentially for both $M(\textit{true})^p$ and $M(\textit{false})^p$. In contrast to the pure three-valued M, the parameterised $M(x_1)$ captures the fact that the transition values of (s_0, s_1) and (s_0, s_2) are unknown but also *complementary*, which gives us the necessary precision for a definite result. The gain of precision comes at the cost of an increase of the number of instantiations *exponential* in the number of parameters. Each instantiation corresponds to a distinct model checking problem that has to be solved. Subsequently, we will show that in *iterative* parameterisation, both the *effort for solving each instance* and the *number of instances to be considered* can be significantly reduced.

3 PMC via Path Tracking

Rule-based parameterisation [12] is an alternative to classical refinement: Instead of adding predicates to an abstract model, the value of selected transitions and predicates is changed from $\perp$ to logical expressions over a parameter set X_n. Such a parameterisation is typically applied iteratively until a definite result in model checking can be obtained, or until classical refinement is inevitable for a definite outcome. In contrast to classical refinement, parameterisation does not affect the state space. If π is a path of a three-valued Kripke structure

M, then π is also a path or at least corresponds to a sequence of states in any parameterisation $M(X_n)$ and its possible instantiations. For instance, the path $\pi = s_0 s_2 s_2 \ldots$ from the Kripke structure M in our running example from Section 2 also exists in the parameterised Kripke structure $M(x_1)$. Moreover, this π is a path in the instantiation $M(\mathit{true})$ and it corresponds to a sequence of states in $M(\mathit{false})$. This allows us consider the same π in the context of different Kripke structures. In terms of LTL properties this means that we can evaluate a formula ψ on a particular path resp. sequence π in a Kripke structure M or any parameterisation $M(X_n)$ and its instantiations $M(B_n)$. In order to make clear *which* Kripke structure is currently in use, we refer to the evaluation of ψ on π in M by $[\pi \models \psi]_M$. The semantics of this evaluation follows from Definition 3.

Now we show that in case of an *unknown* result for a 3MC problem $[M \models_E \psi]$, paths discovered in M can be exploited for solving any corresponding parameterised problem $[M(X_n) \models_E \psi]$ more efficiently. According to Lemma 1, the outcome of 3MC is *unknown* if the following holds: $[M^p \models_E \psi] = \mathit{false}$ and $[M^o \models_E \psi] = \mathit{true}$. Hence, there exist paths in the optimistic completion M^o that witness the property ψ, but no such path exists in the pessimistic completion M^p. The optimistic completion is an over-approximation of M in terms of the existential validity of LTL$^+$ properties. This over-approximation relation also holds between M^o and any instantiation $M(B_n)^y$, $y \in \{o, p\}$, of a parameterisation $M(X_n)$ of M. With regard to paths we get the following lemma:

Lemma 2. *Let $M(X_n)$ be an arbitrary parameterisation of a three-valued Kripke structure M over AP, and let ψ be an LTL$^+$ formula over AP. Then for any $B_n \in \{\mathit{true}, \mathit{false}\}^n$ and any $y \in \{o, p\}$ the following holds:*

$$\forall \pi \in \Pi_{M(B_n)^y}\ [\pi \models \psi]_{M(B_n)^y} = \mathit{true} \ \Rightarrow\ [\pi \models \psi]_{M^o} = \mathit{true}$$

Proof. See `http://www.cs.up.ac.za/cs/ntimm/proofLemma2.pdf`

Hence, each path π that witnesses a property ψ in any instantiation of a parameterisation of M is also a witness for ψ in the optimistic completion M^o. The set $\Pi^{\psi}_{M^o} = \{\pi \in \Pi_{M^o} \mid [\pi \models \psi]_{M^o} = \mathit{true}\}$ is thus a superset of the set of all paths π with $[\pi \models \psi]_{M(B_n)^y} = \mathit{true}$ for any $M(B_n)^y$, $y \in \{o, p\}$. This fact can be exploited to reduce the effort for performing PMC with iterative parameterisation. Solving a parameterised model checking problem $[M(X_n) \models_E \psi]$ requires to determine whether for each instantiation $M(B_n)$ there exists a path π with $[\pi \models \psi]_{M(B_n)^p} = \mathit{true}$, or whether for all paths $[\pi \models \psi]_{M(B_n)^o} = \mathit{false}$ holds. In general, this involves the exhaustive exploration of the state space of each instantiation. Provided that we already have computed the set $\Pi^{\psi}_{M^o}$ we now can solve $[M(X_n) \models_E \psi]$ (and any other parameterisation of the original problem) by just iterating over the paths in $\Pi^{\psi}_{M^o}$ due to of the aforementioned superset relation. Lemma 2 together with Definition 6 gives us the following theorem:

Theorem 1. *Let $M(X_n)$ be the parameterisation of a three-valued Kripke structure M over AP, and let ψ be an LTL$^+$ formula over AP. Furthermore, let $\Pi^{\psi}_{M^o} = \{\pi \in \Pi_{M^o} \mid [\pi \models \psi]_{M^o} = \mathit{true}\}$. Then the following holds:*

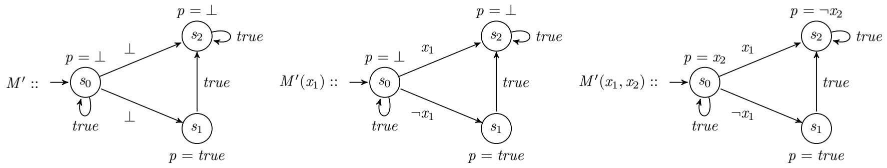
$$[M(X_n) \models_E \psi] = \begin{cases} \mathit{true} & \mathit{iff}\ \forall\, B_n \in \{t,f\}^n\ \exists\, \pi \in \Pi^{\psi}_{M^o}\ [\pi \models \psi]_{M(B_n)^p} = \mathit{true} \\ \mathit{false} & \mathit{iff}\ \forall\, B_n \in \{t,f\}^n\ \forall\, \pi \in \Pi^{\psi}_{M^o}\ [\pi \models \psi]_{M(B_n)^o} = \mathit{false} \\ \bot & \mathit{else} \end{cases}$$

Thus, instead of conducting model checking from scratch for each possible instantiation of $M(X_n)$, we only need to track the paths from $\Pi^{\psi}_{M^o}$ on each instantiation. For our running example we have already seen that model checking the initially non-parameterised problem $[M \models_E \mathbf{F}p]$ yields *unknown*. We now compute the set of witness paths for $\mathbf{F}p$ in the optimistic completion M^o, which is $\Pi^{\mathbf{F}p}_{M^o} = \{\pi_1 = (s_0 s_1 s_2 \ldots), \pi_2 = (s_0 s_2 \ldots)\}$. Due to Lemma 2 we have that $\Pi^{\mathbf{F}p}_{M^o}$ is a superset of all paths π with $[\pi \models \mathbf{F}p]$ for each (optimistic or pessimistic) instantiation of any parameterisation of M. The parameterised model $M(x_1)$ from our running example has the two possible pessimistic instantiations $M(\mathit{true})^p$ and $M(\mathit{false})^p$. By tracking the paths from $\Pi^{\mathbf{F}p}_{M^o}$ we can show that in both instantiations there exists a path that witnesses $\mathbf{F}p$: $[\pi_1 \models \mathbf{F}p]_{M(\mathit{false})^p} = \mathit{true}$ and $[\pi_2 \models \mathbf{F}p]_{M(\mathit{true})^p} = \mathit{true}$. Now Theorem 1 allows us to conclude that $[M(x_1) \models_E \mathbf{F}p] = \mathit{true}$. We have solved a *parameterised* three-valued model checking problem by just tracking witness paths previously discovered in the corresponding *pure* three-valued model.

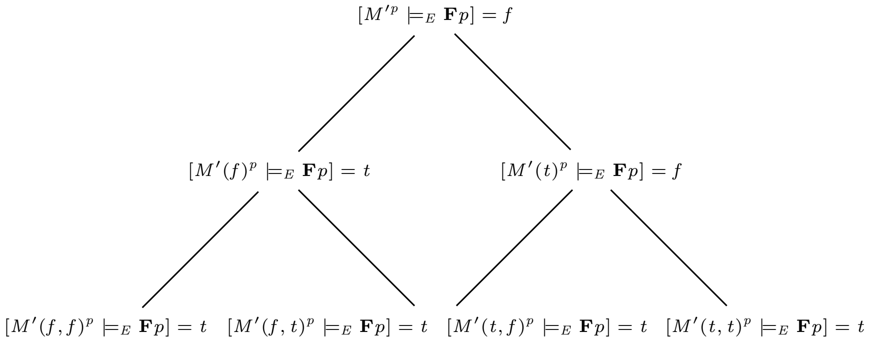
Iterative parameterisation of an initially non-parameterised problem thus allows to exploit information obtained in past iterations in order to reduce the search space for witness paths in future iterations. The computational savings of conducting PMC via *path tracking* will become evident when PMC is reduced to propositional logic satisfiability, which we discuss from Section 5 on.

4 Instantiation Reduction in Iterative Parameterisation

The basic concept of PMC is to verify *each* possible instantiation of a parameterised model and to check whether the results are consistent, i.e. *true* for all pessimistic instances or *false* for all optimistic ones. Inconsistency, i.e. some *true* and some *false* results, may be ruled out via further parameterisation, which however leads to an exponential increase in the number of instantiations. Here we show that in *iterative* parameterisation not necessarily all of these instantiations have to be considered. Partial results from past iterations can be used to reduce the number of relevant instantiations in the current iteration. We illustrate such a reduction based on another example: the three-valued Kripke structure M' and its parameterisations $M'(x_1)$ and $M'(x_1, x_2)$ depicted below.



Checking the property $\mathbf{F}p$ yields the following results: $[M' \models_E \mathbf{F}p] = \bot$, $[M'(x_1) \models_E \mathbf{F}p] = \bot$ and $[M'(x_1, x_2) \models_E \mathbf{F}p] = \mathit{true}$. Hence, the iteration that introduced the second parameter[1] x_2 has brought the necessary precision for a definite result in verification. The following tree characterises the iterations and the pessimistic instantiations[2] that have to be considered in order to prove that $\mathbf{F}p$ holds for our example problem with iterative parameterisation:


$[M'^p \models_E \mathbf{F}p] = f$
$[M'(f)^p \models_E \mathbf{F}p] = t$
$[M'(t)^p \models_E \mathbf{F}p] = f$
$[M'(f,f)^p \models_E \mathbf{F}p] = t$
$[M'(f,t)^p \models_E \mathbf{F}p] = t$
$[M'(t,f)^p \models_E \mathbf{F}p] = t$
$[M'(t,t)^p \models_E \mathbf{F}p] = t$

In the third iteration we get consistently *true* results for all possible instantiations of $M'(x_1, x_2)^p$, which allows us to conclude that $\mathbf{F}p$ holds. However, checking *all* instances is actually not necessary under iterative parameterisation. Note that we obtained the partial result $[M'(\mathit{false})^p \models_E \mathbf{F}p] = \mathit{true}$ in the previous iteration. It is easy to show that any instantiation of a *further* parameterisation of $M'(\mathit{false})$ satisfies equally many or more LTL^+ properties than the pessimistic instantiation $M'(\mathit{false})^p$. The argument here is that parameterisation always involves a replacement of selected $\perp$'s by parameter expressions. While these $\perp$'s would be set to *false* under a pessimistic instantiation, the replacing parameter expressions might also take the value *true* in an instantiation of a further parameterised model, which generally leads to the satisfaction of *more* LTL^+ properties. Thus, for our running example we can immediately conclude

$$[M'(\mathit{false})^p \models_E \mathbf{F}p] = \mathit{true} \quad \Rightarrow \quad \wedge \begin{array}{l} [M'(\mathit{false}, \mathit{false})^p \models_E \mathbf{F}p] = \mathit{true} \\ [M'(\mathit{false}, \mathit{true})^p \models_E \mathbf{F}p] = \mathit{true} \end{array}$$

which saves us the explicit consideration of the instantiations $M'(\mathit{false}, \mathit{false})^p$ and $M'(\mathit{false}, \mathit{true})^p$ and thus significantly reduces the computational effort for solving our example PMC problem. Lemma 3 generalises the capabilities for *instantiation reduction* in iterative parameterisation:

Lemma 3. *Let $M(B_n)$ be an instantiation of a parameterised three-valued Kripke structure $M(X_n)$ and let $M(B_n)(X_m)$ be a further parameterisation of $M(B_n)$. Then for any $\psi \in LTL^+$ the following holds:*

[1] This parameterisation step indicates that the value of the unknown predicate p is *inverted* by taking the transition from s_0 to s_2, which might be a fact automatically derived from the modelled system via our parameterisation rules defined in [12].

[2] The *optimistic* instantiations that are considered for attempting to *disprove* the property yield a similar tree.

1. $[M(B_n)^p \models_E \psi] = \mathit{true} \;\Rightarrow\; \forall\, B_m \in \{t,f\}^m \;\; [M(B_n)(B_m)^p \models_E \psi] = \mathit{true}$

2. $[M(B_n)^o \models_E \psi] = \mathit{false} \;\Rightarrow\; \forall\, B_m \in \{t,f\}^m \;\; [M(B_n)(B_m)^o \models_E \psi] = \mathit{false}$

Proof. See `http://www.cs.up.ac.za/cs/ntimm/proofLemma3.pdf`

This lemma establishes the useful relation between results obtained for an instantiation $M(B_n)^y$, $y \in \{o,p\}$, and for any 'successor' instantiation $M(B_n)(B_m)^y$, that allows us to cut down the number of instantiations to be considered in future iterations. In our iterative approach to PMC we thus memorise cases where a *true* result for a *pessimistic* instantiation or a *false* result for an *optimistic* instantiation was obtained. In subsequent iterations the consideration of instances that originate from the further parameterisation of these cases is no longer necessary. Lemma 3 together with Definition 6 gives us the following theorem:

Theorem 2. *Let $M(X_n) = (S, s_0, R(X_n), L(X_n))$ be a parameterised three-valued Kripke structure over AP and $X = (x_1, \ldots, x_n)$. Moreover, let ψ be an LTL$^+$ formula over AP. Then the following holds:*

$$[M(X_n) \models_E \psi] = \begin{cases} \mathit{true} \;\; \mathit{iff}\; \forall (B_n) \in \{t,f\}^n \; \boxed{\mathit{with} \;\; \neg\, \exists\, m \leq n : [M(B_{n-m})^p \models_E \psi] = t} : \\ \qquad\quad ([M(B_n)^p \models_E \psi] = \mathit{true}) \\ \mathit{false} \;\; \mathit{iff}\; \forall (B_n) \in \{t,f\}^n \; \boxed{\mathit{with} \;\; \neg\, \exists\, m \leq n : [M(B_{n-m})^o \models_E \psi] = f} : \\ \qquad\quad ([M(B_n)^o \models_E \psi] = \mathit{false}) \\ \bot \qquad \mathit{else} \end{cases}$$

The (boxed) constraints on the number of parameter instantiations that need to be considered in order to solve a PMC problem allow to perform PMC with iterative parameterisation with significantly less computational effort. For our running example we were e.g. able to reduce the number of relevant instantiations in the third iteration *by half.* Theorem 2 can be straightforwardly combined with Theorem 1 which enables us to profit from *instantiation reduction* and *path tracking* at the same time. In the following sections we will show that our enhancements of PMC can be transferred to *SAT-based* PMC. We start with a brief review of the basics of SAT-based parameterised three-valued model checking.

5 Background: SAT-Based Bounded PMC

In [13] we showed that a PMC problem can be encoded as a parameterised propositional logic formula and then solved by applying SAT solving to each formula instance, which we briefly review here. A prerequisite is to *bound* the length of the considered paths by some $k \in \mathbb{N}$. By Π_{M_k} we denote the set of all *k-bounded* paths in a Kripke structure M. Such finite prefixes $\pi(0) \ldots \pi(k)$ can still represent infinite paths if the prefix has a *k-loop*, i.e. the last state $\pi(k)$

has a successor state that is also part of the prefix. For the bounded evaluation of *LTL*$^+$ properties we have to distinguish between paths *with* and *without* a k-loop.

Definition 7 (Three-Valued Bounded Evaluation of LTL$^+$). *Let $M = (S, s_0, R, L)$ over AP be a complement-closed three-valued Kripke structure, let $k \in \mathbb{N}$ and let π be a path of M without a k-loop. Then the k-bounded evaluation of an LTL$^+$ formula ψ on π, written $\left[\pi \models_k^i \psi\right]$ where $i \in \mathbb{N}$ with $i \leq k$ denotes the current position along the path, is inductively defined as follows*

$$\begin{array}{lll}
\left[\pi \models_k^i p\right] & := & L(\pi(i), p) \\
\left[\pi \models_k^i \psi \vee \psi'\right] & := & \left[\pi \models_k^i \psi\right] \ \vee \ \left[\pi \models_k^i \psi'\right] \\
\left[\pi \models_k^i \mathbf{G}\psi\right] & := & \mathit{false} \\
\left[\pi \models_k^i \mathbf{F}\psi\right] & := & \bigvee_{j=i}^{k}([\pi \models_k^j \psi] \wedge \bigwedge_{l=i}^{j-1} R(\pi(l), \pi(l+1)))
\end{array}$$

If π has a k-loop, then $[\pi \models_k^i \psi] := [\pi^i \models \psi]$. Moreover, the existential value of ψ in M under the bounded semantics is $[M \models_{E,k} \psi] := \bigvee_{\pi \in \Pi_{M_k}} [\pi \models_k^0 \psi]$.

If all $k \in \mathbb{N}$ are considered then the bounded semantics is equivalent to the unbounded one: $[M \models_E \psi] = \bigvee_{k \in \mathbb{N}} [M \models_{E,k} \psi]$. Typically, the bound is iteratively increased until the property of interest can be either proven or a completeness threshold is reached. The bounded semantics for 3MC can be extended to PMC:

Definition 8 (Bounded Parameterised Three-Valued Model Checking). *Let $M(X_n) = (S, s_0, R(X_n), L(X_n))$ be a parameterised three-valued KS over AP and X_n. Let ψ be an LTL$^+$ formula over AP and $k \in \mathbb{N}$. The existential value of ψ in $M(X_n)$ under the bounded semantics is*

$$[M(X_n) \models_{E,k} \psi] := \begin{cases} \mathit{true} & \mathit{if} \ \forall\, B_n \in \{t, f\}^n \ ([M(B_n)^p \models_{E,k} \psi] = \mathit{true}) \\ \mathit{false} & \mathit{if} \ \forall\, B_n \in \{t, f\}^n \ ([M(B_n)^o \models_{E,k} \psi] = \mathit{false}) \\ \perp & \mathit{else} \end{cases}$$

As shown in [13] bounded PMC can be reduced to the new satisfiability problem SAT$_{X3}$. For a parameterised three-valued Kripke structure $M(X_n)$ over AP and X_n, an LTL$^+$ formula ψ and a bound $k \in \mathbb{N}$, a parameterised propositional logic formula $F(X_n)_k$ is constructed such that $[M(X_n) \models_{E,k} \psi] = \text{SAT}_{X3}(F(X_n)_k)$.

Solving SAT$_{X3}$ reduces to solving classical SAT for each possible parameter instantiation. We briefly review how $F(X_n)_k$ is constructed for a given PMC problem. The construction of $F(X_n)_k$ is divided into the translation of $M(X_n)$ into a formula $[\![M(X_n)]\!]_k$ and the translation of ψ into a formula $[\![\psi]\!]_k$. The encoding of $M(X_n)$ first requires to encode its states as Boolean formulae. For this, a set of Boolean atoms $\{A, B, \ldots\}$ is introduced. Let PL be the set of propositional formulae over $\{A, B, \ldots\}$ and the constants *true* and *false.* Then an encoding of the states of a Kripke structure is defined as follows.

Definition 9 (State Encoding). *Let $M(X_n) = (S, s_0, R(X_n), L(X_n))$ be a parameterised Kripke structure. A Boolean encoding of its states corresponds to an injective mapping $e : S \to PL$ where $\forall\, s \in S : e(s)$ is a conjunction of literals.*

$M(X_n)$ can now be translated into a formula $[\![M(X_n)]\!]_k$ that characterises k-bounded paths in $M(X_n)$. As we consider states as parts of such paths, the state encoding is extended by index values $i \in \{0, ..., k\}$ where i denotes the position along a path. We get an extended set of indexed atoms $\{A_0, B_0, ..., A_k, B_k, ...\}$.

Definition 10 (Kripke Structure Encoding). *Let $M(X_n) = (S, s_0, R(X_n), L(X_n))$ be a parameterised three-valued KS and e an encoding of its states. We define $Init_0$ as the predicate characterising the initial state of $M(X_n)$ with*

$$Init_0 := e(s_0)_0$$

and $T_{i,i+1}$ as the predicate that characterises the transitions of $M(X_n)$ with

$$T_{i,i+1} := \bigvee_{s \in S} \bigvee_{s' \in S} e(s)_i \wedge e(s')_{i+1} \wedge R(X_n)(s, s').$$

Then the entire encoding of $M(X_n)$ for a bound $k \in \mathbb{N}$ is defined as

$$[\![M(X_n)]\!]_k := Init_0 \wedge \bigwedge_{i=0}^{k-1} T_{i,i+1}$$

The second part of the encoding concerns the LTL^+ formula ψ. This generally requires to distinguish the cases where ψ is evaluated on a path *with* and *without* a loop. Due to space limitations we only consider the case without a loop here.

Definition 11 (LTL$^+$ Encoding without Loop). *Let p be an atomic predicate, ψ and ψ' LTL^+ formulae, and $k, i \in \mathbb{N}$ with $i \leq k$.*

$$\begin{array}{lcl@{\qquad\qquad}lcl} [\![p]\!]_k^i & := & \bigvee_{s \in S} e(s)_i \wedge L(s,p) & [\![\mathbf{G}\psi]\!]_k^i & := & false \\ [\![\psi \vee \psi']\!]_k^i & := & [\![\psi]\!]_k^i \vee [\![\psi']\!]_k^i & [\![\mathbf{F}\psi]\!]_k^i & := & \bigvee_{j=i}^{k} [\![\psi]\!]_k^j \end{array}$$

The overall encoding of $[M(X_n) \models_{E,k} \psi]$ is now $F(X_n)_k := [\![M(X_n)]\!]_k \wedge [\![\psi]\!]_k^0$. $F(X_n)_k$ is defined over atoms but also over X_n and $\perp$. Thus, SAT_{X3} is not a standard satisfiability problem. It reduces to multiple instances of classical SAT.

$$\text{SAT}_{X3}(F(X_n)_k) := \begin{cases} true & if \;\; \forall\, B_n \in \{t, f\}^n \; (\text{SAT}(F(B_n)_k^p) = true) \\ false & if \qquad\qquad\qquad\quad\;\; \text{SAT}(F(X_n)_k^o) = false \\ \perp & else \end{cases}$$

where $F(B_n)_k^p = F(X_n)_k[X_n/B_n][\perp/false]$ and $F(X_n)_k^o = F(X_n)_k[\perp/true]$. Since $\text{SAT}(F((X_n)_k^o) = f$ is equivalent to $\forall\, B_n \in \{t, f\}^n$ $(\text{SAT}(F((B_n)_k^o) = f)$, checking whether SAT_{X3} yields *false* requires a single SAT test only. The result of the SAT_{X3} test is equivalent to the result of the encoded PMC problem [13]:

Lemma 4. *Let $M(X_n)$ be a parameterised three-valued Kripke structure over a set of predicates AP, let ψ be an LTL^+ formula over AP, and $k \in \mathbb{N}$. Then*

$$[M(X_n) \models_{E,k} \psi] \;=\; \text{SAT}_{X3}(F(X_n)_k)$$

Thus, bounded PMC can be reduced to multiple instances of SAT. Subsequently, we show that in iterative parameterisation, solving these instances can be efficiently performed via *interpretation validation* in place of SAT solving.

6 Bounded PMC via Interpretation Validation

As we have seen, bounded PMC reduces to multiple instances of classical SAT solving. Given a propositional logic formula F over a set of Boolean atoms $\mathcal{A}$, SAT is the (NP-complete) problem of determining whether there exists an interpretation $I : \mathcal{A} \rightarrow \{\mathit{true}, \mathit{false}\}$ that satisfies F. The previously introduced encoding $F(X_n)_k$ of a bounded PMC problem $[M(X_n) \models_{E,k} \psi]$ has the nice feature that each interpretation that satisfies an instantiation of the encoding exactly characterises a path in the corresponding model that witnesses the property ψ within k steps [12]. Now we will show that, based on this feature, the solving performance of SAT-based PMC with iterative parameterisation can be considerably improved. De facto, we transfer our concept *path tracking* in explicit (non-encoded) PMC (Section 3) to the SAT scenario. Here the satisfiability tests associated with an encoded problem can be replaced by the significantly less expensive *validation* of satisfying interpretations discovered in earlier iterations.

For illustration, we consider again the pure three-valued Kripke structure M and its parameterisation $M(x_1)$ from Section 2. For the states of M we choose the encoding $e(s_0)_i = \neg A_i \wedge \neg B_i$, $e(s_1)_i = \neg A_i \wedge B_i$ and $e(s_2)_i = A_i \wedge \neg B_i$ over the set of atoms $\mathcal{A} = \{A_0, B_0, \dots, A_k, B_k\}$. The bounded model checking problem $[M \models_{E,k} \mathbf{F}p]$ can now be logically encoded as follows

$$\begin{aligned}
F_k = \;& \mathit{Init}_0 \wedge \textstyle\bigwedge_{i=0}^{k-1} T_{i,i+1} \wedge [\![\mathbf{F}p]\!]_k^0 \\
= \;& (\neg A_0 \wedge \neg B_0) \wedge \textstyle\bigwedge_{i=0}^{k-1} ((\neg A_i \wedge \neg B_i \wedge \neg A_{i+1} \wedge \neg B_{i+1} \wedge \mathit{true}) \\
& \vee (\neg A_i \wedge \neg B_i \wedge \neg A_{i+1} \wedge B_{i+1} \wedge \bot_{(s_0,s_1)}) \vee (\neg A_i \wedge \neg B_i \wedge A_{i+1} \wedge \neg B_{i+1} \wedge \bot_{(s_0,s_2)}) \\
& \vee (\neg A_i \wedge B_i \wedge A_{i+1} \wedge \neg B_{i+1} \wedge \mathit{true}) \vee (A_i \wedge \neg B_i \wedge A_{i+1} \wedge \neg B_{i+1} \wedge \mathit{true})) \\
& \wedge \textstyle\bigvee_{i=0}^{k} ((\neg A_i \wedge \neg B_i \wedge \mathit{false}) \vee (\neg A_i \wedge B_i \wedge \mathit{false}) \vee (A_i \wedge \neg B_i \wedge \mathit{true}))
\end{aligned}$$

Note that we annotated the $\bot$'s in F_k with the transitions in M they are associated with, which helps to distinguish individual $\bot$'s in the later parameterisation. Since M is *pure* three-valued, SAT_{X3} reduces to two satisfiability instances $F_k^p = F_k[\bot/\mathit{false}]$ and $F_k^o = F_k[\bot/\mathit{true}]$. F_k^p is the propositional logic equivalent of the explicit PMC problem $[M^p \models_{E,k} \mathbf{F}p]$, whereas F_k^o is the equivalent of $[M^o \models_{E,k} \mathbf{F}p]$. We get $\mathrm{SAT}(F_k^p) = \mathit{false}$ and $\mathrm{SAT}(F_k^o) = \mathit{true}$, which gives us *unknown* as the overall result. Hence, there exist interpretations that satisfy F_k^o but there is no interpretation that satisfies F_k^p. Several approaches for the efficient computation of *all* satisfying interpretations of a propositional logic formula have been introduced [8, 15, 14], which we will discuss in detail in the related work section. These approaches for solving the so-called AllSAT problem allow us to compute the set $\mathcal{I}_{F_k^o} = \{I \mid I(F_k^o) = \mathit{true}\}$ of all interpretations that make F_k^o *true*. As stated before, each $I \in \mathcal{I}_{F_k^o}$ exactly characterises a k-bounded path π with $[\pi \models_k^0 \mathbf{F}p]_{M^o} = \mathit{true}$. For instance, the interpretation $I : A_0 \mapsto \mathit{false}, B_0 \mapsto \mathit{false}, A_1 \mapsto \mathit{false}, B_1 \mapsto \mathit{true}, A_2 \mapsto \mathit{true}, B_2 \mapsto \mathit{false}$ for a 2-bounded encoding F_2 describes the path $\pi = s_0 s_1 s_2$ in M. Since π contains the *unknown* transition (s_0, s_1) we call it an *unconfirmed witness* for the property of interest. Based on such unconfirmed witnesses the parameterisation rules we defined in [12] can be automatically and iteratively applied, which, in this particular case, gives us the parameterised Kripke structure $M(x_1)$ from our running example. This parameterisation step can be straightforwardly lifted to the level of the propositional encoding. Based on an analysis of the interpretations in $\mathcal{I}_{F_2^o}$ the current encoding can be parameterised to $F(x_1)_2 := F_2[\bot_{(s_0,s_1)}/\neg x_1][\bot_{(s_0,s_2)}/x_1]$.

Solving $\mathrm{SAT}_{X3}(F(X_n)_k)$ generally requires to solve SAT *from scratch* for each instantiation of $F(X_n)_k$. However, since each interpretation satisfying the non-parameterised F_k^o characterises a path π with $[\pi \models_k^0 \psi]_{M^o} = \mathit{true}$ and vice versa, we also have a one-to-one correspondence between the elements of the set of interpretations $\mathcal{I}_{F_k^o}$ and the set of paths $\Pi_{M_k^o}^{\psi}$ (compare Section 3). In accordance with the results from Section 3, $\mathcal{I}_{F_k^o}$ is a superset of any $\mathcal{I}_{F(B_n)_k^y}$ where $F(B_n)_k = F(X_n)_k[x_1,\ldots,x_n/b_1,\ldots,b_n]$ is an instantiation of an arbitrary parameterisation $F(X_n)_k$ of F_k and $y \in \{o,p\}$. In an iterative approach, this allows us to solve encoded PMC problems via *interpretation validation.* Theorem 1 together with Lemma 4 gives us the following theorem:

Theorem 3. *Let* $[M(X_n) \models_{E,k} \psi]$ *be a parameterisation of a bounded 3MC problem* $[M \models_{E,k} \psi]$*. Moreover, let* F_k *and* $F(X_n)_k$ *be the corresponding propositional logic encodings and* $\mathcal{I}_{F_k^o} = \{I \mid I(F_k^o) = \mathit{true}\}$*. Then:*

$$[M(X_n) \models_{E,k} \psi] = \begin{cases} \mathit{true} & \mathit{iff}\ \forall B_n \in \{t,f\}^n\ \exists I \in \mathcal{I}_{F_k^o} : I(F(B_n)_k^p) = \mathit{true} \\ \mathit{false} & \mathit{iff}\ \qquad\qquad\qquad\ \forall I \in \mathcal{I}_{F_k^o} : I(F(X_n)_k^o) = \mathit{false} \\ \bot & \mathit{else} \end{cases}$$

Hence, instead of solving one NP-complete SAT problem per instance we just need to validate $\left|\mathcal{I}_{F_k^o}\right|$ interpretations per instance, which requires linear time and space per interpretation. For the optimistic completion F_2^o of our initially non-parameterised example encoding we can efficiently compute the set $\mathcal{I}_{F_2^o}$ via AllSAT methods. $\mathcal{I}_{F_2^o}$ consists of three interpretations I_1, I_2 and I_3 characterising the bounded paths $\pi_1 = s_0s_1s_2$, $\pi_2 = s_0s_0s_2$ and $\pi_3 = s_0s_2s_2$. In the next iteration we consider each instantiation of the parameterised encoding $F(x_1)_2$ and validate the interpretations: I_1 satisfies $F(\mathit{false})_2^p$ and I_2 satisfies $F(\mathit{true})_2^p$. Thus, we can conclude that $[M(x_1) \models_{E,2} \mathbf{F}p] = \mathit{true}$. Bounded PMC via *interpretation validation* is straightforwardly compatible with *instantiation reduction* (compare Section 4). In the next section we introduce our model checking framework for PMC problems that implements our proposed enhancements.

7 PMC Framework with Iterative Parameterisation

We have developed a prototype (All)SAT-based bounded model checker for PMC problems. It employs the solver library Sat4j [10] and implements the algorithm for efficient AllSAT solving proposed in [15]. The checker iterates over the bound k, starting with $k = 0$, and over possible parameterisations of the initially non-parameterised input model. In each iteration that yields *unknown* it is checked whether any parameterisation rule from [12] is applicable. If further parameterisation is currently not feasible the bound is incremented. Our checker supports two modes: In the *basic* mode in each iteration all instances of the encoded problem are processed via *incremental SAT solving* [6]: For acceleration, learned conflict clauses are reused between the solvers processing different SAT instances, in case the learning happened based on a common subformula. No instantiation reduction is applied in the basic mode. The *enhanced* mode works as follows: After every bound incrementation the AllSAT problem for the propositional logic encoding F^o_k is solved[3], which gives us the set $\mathcal{I}_{F^o_k}$ of all satisfying interpretations. Between the AllSAT problems for different bounds we also employ incremental solving, i.e. we reuse learned clauses. After every parameterisation step we apply *interpretation validation* based on $\mathcal{I}_{F^o_k}$ for the instantiations of the parameterised encoding $F(X_n)_k$. Via *instantiation reduction* we additionally narrow the number of instantiations that actually have to be considered. The following procedure characterises a single iteration and illustrates how our enhancements have been implemented:

```
Data: current $F(X_n)^p_k$ and $\mathcal{I}_{F^o_k}$
for all $B_n \in \{true, false\}^n$ do
    if $SATResult(F(B_{n-1})^p_k) = true$ then
        $SATResult(F(B_n)^p_k) := true$
    else
        $SATResult(F(B_n)^p_k) := validate(\mathcal{I}_{F^o_k}, F(B_n)^p_k)$
```

Here *SATResult* is the database where we store satisfiability results. It allows us to look up results from past iterations: If $SATResult(F(B_{n-1})^p_k) = true$ then we can skip the computation of the SAT result for $F(B_n)^p_k$ and immediately conclude that it is *true* as well, which implements *instantiation reduction.* The function call $validate(\mathcal{I}_{F^o_k}, F(B_n)^p_k)$ returns whether any interpretation from $\mathcal{I}_{F^o_k}$ is satisfying for $F(B_n)^p_k$, and thus, implements *interpretation validation.* We use an analogous procedure for processing $F(X_n)^o_k$ where *false* results from past iterations are exploited in a similar way.

In preliminary experiments we compared the run-time of PMC in the *basic* and the *enhanced* mode. We considered a number of initially non-parameterised problems that we encoded in propositional logic and then iteratively parameterised until a definite result could be obtained. In the enhanced mode we achieved substantial savings in verification time. While a more extensive experimental evaluation is in preparation, we so far observed that the actual performance gain due to the application of interpretation validation instead of SAT solving results from the trade-off between the extra costs of solving AllSAT for the initial problem and the savings in the later iterations: The more iterations were needed, the more we generally saved in the enhanced mode. And, the smaller the set $\mathcal{I}_{F^o_k}$, the more we profited from interpretation validation in PMC. – In terms of a potential reduction of the *size* of $\mathcal{I}_{F^o_k}$, we can actually derive another beneficial fact based on Lemma 3 and the encoding definitions: If an $I \in \mathcal{I}_{F^o_k}$ does not satisfy *any* instantiation in the current iteration then it will also not satisfy any instantiation in a future iteration. This allows to immediately remove such an interpretation from $\mathcal{I}_{F^o_k}$, which involves further computational savings.

[3] The encoding has been transferred into conjunctive normal form via Tseitin transformation. Thus, we only consider interpretations that differ wrt. the valuation of the original problem variables, and not wrt. newly introduced auxiliary variables.

8 Related Work

Iterative verification approaches with *reuse of results* have been considered in several ways. An established approach is *counterexample-guided abstraction refinement* (CEGAR) [4]. The CEGAR-based model checker Yasm [9] reuses results between abstraction iterations as follows: Each iteration involves the computation of the set of states from which an error can be definitely reached. Instead of checking the error reachability from scratch in a subsequent iteration, Yasm just checks whether any state from the previously computed set is reachable. Another common approach is SAT-based *incremental bounded model checking* [6]. Each iteration corresponds to solving a SAT instance that encodes the model checking problem for a different bound. Results are reused in the sense of sharing conflict clauses that have been learned based on common parts of the SAT instances. While these approaches iterate over *predicate abstractions* and *bounds*, our approach is based on iterative *parameterisation.* Each iteration involves the consideration of all instantiations of a parameterised model. A related approach is *conditional model checking* (CMC) [1]. Here the property is checked under different conditions that restrict *which* part of the model is explored. Multiple checks under complementary conditions allow to obtain an 'unconditional' result. CMC has so far not been considered in an iterative context. In our PMC approach we do not check *different parts* of a model but *different approximations* (parameter instantiations) of the system to be verified, that together yield an exact result. Our approach relies on AllSAT, for which several memory and time efficient solving techniques have been proposed. The technique from [15] uses *blocking clauses* to avoid that the same interpretations are found again. It is based on an *incremental* search for interpretations that prevents starting search from scratch after an interpretation was found. Moreover, generated blocking clauses are *merged*, which improves memory efficiency. Other AllSAT solving techniques that follow similar concepts have been introduced in [8, 14].

9 Conclusion and Outlook

We introduced an *iterative* approach to SAT-based parameterised three-valued model checking that substantially overcomes the former drawbacks of parameterisation in terms of computational costs. Our new concept *instantiation reduction* allows to exploit results from past iterations in order to narrow the *number* of instantiations that have to be considered in future iterations. With *interpretation validation* we introduced a concept that significantly reduces the *effort* for solving the individual instantiations occurring in PMC. We proved the soundness of our PMC-improving concepts and integrated them into a prototype model checker. Preliminary experiments revealed that our approach leads to a practically relevant speed-up of PMC. It allows to profit from the extra precision of parameterisation without suffering from the previous performance drawbacks.

We are currently working on a concept for instantiation reduction based on results obtained for *different bounds*: Partial results for the instances of an encoding $F(X_n)_k$ can be also exploited for further reducing the number of relevant instances of all subsequent encodings $F(X_{n+m})_{k+l}$. Another direction for future research is the development of a concept for *summarising* interpretations in $\mathcal{I}_{F_k^o}$ into equivalence classes: For interpretations that characterise paths that are bisimilar or stuttering equivalent wrt. the property of interest it is sufficient to have just one representative in $\mathcal{I}_{F_k^o}$, which allows further computational savings due to interpretation validation. We also plan to combine our iterative framework with a classical abstraction refinement procedure: In case parameterisation and bound incrementation do not yield the necessary precision for a definite result, more predicates can be added to the abstract model. *AllSAT-based Predicate abstraction and refinement* [5] can then work hand-in-hand with our AllSAT approach. Finally, the summarisation of all SAT instances occurring per iteration to a single *quantified* Boolean formula (QBF) and the subsequent application of QBF interpretation validation is another direction for future investigation.

References

1. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proceedings of the ACM SIGSOFT FSE '12. pp. 57:1–57:11. ACM, New York (2012)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Handbook of Satisfiability 185, 457–481 (2009)
3. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: CAV 1999. pp. 274–287. LNCS, Springer Berlin Heidelberg (1999)
4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV, LNCS, vol. 1855, pp. 154–169. Springer (2000)
5. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Satabs: Sat-based predicate abstraction for ansi-c. In: TACAS, LNCS, vol. 3440, pp. 570–574. Springer (2005)
6. Eén, N., Sörensson, N.: Temporal induction by incremental sat solving. Electronic Notes in Theoretical Computer Science 89(4), 543–560 (2003)
7. Fitting, M.: Kleene's three valued logics and their children. Fundamenta Informaticae 20(1-3), 113–131 (1994)
8. Grumberg, O., Schuster, A., Yadgar, A.: Memory efficient all-solutions sat solver and its application for reachability analysis. In: Hu, A., Martin, A. (eds.) FMCAD, LNCS, vol. 3312, pp. 275–289. Springer Berlin Heidelberg (2004)
9. Gurfinkel, A., Wei, O., Chechik, M.: Yasm: A software model-checker for verification and refutation. In: CAV, LNCS, vol. 4144, pp. 170–174. Springer (2006)
10. Le Berre, D., Parrain, A.: The sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation 7, 59–64 (2010)
11. Timm, N.: Three-Valued Abstraction and Heuristic-Guided Refinement for Verifying Concurrent Systems. Phd thesis, University of Paderborn (2013)
12. Timm, N., Gruner, S.: Parameterisation of three-valued abstractions. In: Formal Methods: Foundations and Applications, pp. 162–178. Springer (2015)
13. Timm, N., Gruner, S., Sibanda, P.: Parallel sat-based parameterised three-valued model checking. In: SPIN Symposium on Model Checking Software. Springer (2015)
14. Yu, Y., Subramanyan, P., Tsiskaridze, N., Malik, S.: All-sat using minimal blocking clauses. In: VLSI Design 2014. pp. 86–91 (2014)
15. Zhao, W., Wu, W.: Asig: An all-solution sat solver for cnf formulas. In: 11th IEEE International Conference on CAD/Graphics. pp. 508–513 (Aug 2009)

Mobile CSP

Jim Woodcock*, Andy Wellings, and Ana Cavalcanti

Department of Computer Science

University of York

Abstract. We describe an extension of imperative CSP with primitives to declare new event names and to exchange them by message passing between processes. We give examples in Mobile CSP to motivate the language design, and describe its semantic domain, based on the standard failures-divergences model for CSP, but also recording a dynamic event alphabet. The traces component is identical to the separation logic semantics of Hoare & O'Hearn. Our novel contribution is a semantics for mobile channels in CSP, described in Unifying Theories of Programming, that supports: compositionality with other language paradigms; channel faults, nondeterminism, deadlock, and livelock; multi-way synchronisation; and many-to-many channels. We compare and contrast our semantics with other approaches, including the π-calculus, and consider implementation issues. As well as modelling reconfigurable systems, our extension to CSP provides semantics for techniques such as dynamic class-loading and the full use of dynamic dispatching and delegation.

1 Introduction and Overview

Model-driven systems engineering is gaining popularity in large-scale industrial applications; it relies for its success on modelling languages that provide efficient domain-specific abstractions for design, analysis, and implementation. There is no single modelling language that can cover every aspect of a significant system, let alone the complexities of systems of systems or cyber-physical systems. Adequate modelling techniques must inevitably involve heterogeneous semantics, and this raises the scientific question of how to fit these different semantics together: *the model integration problem* [15]. Our approach to understanding the integration of models with diverse semantics is to study the different paradigms in isolation and then find ways of composing them. We are not looking for a unified language to encompass all language paradigms, but rather we seek to unify theories to explain how they fit together and complement each other. This is the research agenda of Unifying Theories of Programming (UTP) [12].

In this paper, we explore the paradigm of reconfigurable systems and programs, where we model interaction between system components (and even between systems themselves) by message passing along channels that form a flexible topology that changes over time. We describe an extension of CSP [11, 22] with

* Corresponding author: `Jim.Woodcock@york.ac.uk`.

primitives to declare new event names and to exchange them in messages over channels. Our semantics is an extended predicative form of the standard failures-divergences semantic model for CSP, enhanced with imperative programming features [12]. The traces component of this model is identical to the concurrent separation logic semantics proposed earlier by Hoare & O'Hearn [13]. The novel contribution of our work is a semantics in UTP that supports the following:

1. Compositionality with other language paradigms. A key feature of UTP is the ability to combine different language features using Galois connections.
2. Formalisation of channel faults, nondeterminism, deadlock, livelock, multi-way synchronisation. and many-to-many channels.

We start the paper in Sect. 2 by recalling the principal features of CSP and of the π-calculus, its process-algebraic cousin with mobile channels. In Sect. 3, we list a series of examples of increasing complexity that display the use of mobile channels in modelling. Having motivated the use of mobility, we define our semantic domain in Unifying Theories of Programming in Sect. 4, and give the semantics of four key operators in Sect. 5: value and channel communications, the creation of new channels, and parallel composition. In Sect. 6, we describe the implementation in Java of an architectural pattern that uses mobile channels. We conclude the paper by describing related and future work in Sects 7 and 8.

2 CSP

CSP is a formal language for describing patterns of interaction in concurrent systems [11, 22]; it is a process algebra based on message passing via channels. It has formed the basis of the following languages: occam, the native programming language for the inmos transputer microprocessors [32], and occam-π, its extension with mobile processes and data [36]; Ada's rendezvous mechanism [14]; JCSP, the CSP library for Java [35]; PyCSP, the CSP library for Python [1]; Scala, the strongly typed functional programming language [20], with its message-passing semantics and Communicating Scala Objects; the *Circus* family of specification and refinement languages [38, 39], including *OhCircus* [5], *SlottedCircus* [3], *CircusTime* [27], and *TravellingCircus* [30, 31]; *CML*, the COMPASS Modelling Language [41, 37]; CSP‖B and Mobile CSP‖B [33, 26, 34]; CSP-OZ and CSP$^{\#}$, stateful, object-oriented versions of the language [10, 29]; rCOS, the component modelling language [16]; and Ptolemy, the embedded systems modelling language [28]. The main elements of CSP are described in Tab. 1.

Different aspects of semantics, such as determinism, nondeterminism, livelock, timing, and fairness, are dealt with in a hierarchy of semantic models, all based on refinement as inverse behavioural inclusion: every implementation behaviour must be specified. A powerful refinement model-checker for CSP, FDR3 [19], supports the language and a basic extension to timed systems.

The π-calculus [18] differs significantly from CSP in permitting channel names to be communicated along the channels themselves, and in this way it is able to describe concurrent computations whose network configurations may change

prefix	$a \rightarrow P$	input	$c?x \rightarrow P(x)$
output	$c!e \rightarrow P$	internal choice	$P \sqcap Q$
external choice	$P \Box Q$	sequence	$P \mathbin{;} Q$
parallel	$P \parallel Q$	abstraction	$P \setminus S$
recursion	$\mu X \bullet F(X)$	deadlock	STOP
termination	SKIP	divergence	CHAOS

Table 1. The main elements of CSP.

during the computation. As well as treating channel names as first-class citizens, the π-calculus has a further primitive, $(\nu x)\, P$, that allows for the creation of a new name allocated as a constant within P. An axiom, known as scope extrusion,

$$(\nu\ x)P \mid Q \;=\; (\nu\ x)(P \mid Q) \qquad\qquad \text{if } x \text{ is not a free name of } Q$$

describes how the scope of a bound name x may be extruded, as would be necessary before an action outputting the name x from P to Q.

In CSP, channel communications are events, and input and output commands are merely abbreviations for choices over event synchronisations:

$$c?x : T \rightarrow P(x) \;\widehat{=}\; \Box\, x : T \bullet c.x \rightarrow P(x) \qquad c!e \rightarrow Q \;\widehat{=}\; c.e \rightarrow Q$$

So a theory of mobile events underpins a theory of mobile channels. In this paper, we propose an extension of imperative CSP with mobile events; this language supports *MobileCircus*, an extension of the *Circus* modelling language. Both language extensions are based on a natural notion of refinement of failures-divergences, which distinguishes them from the π-calculus.

3 Motivation and Examples

One of the main areas underpinned by research in formal methods is software for high-integrity and safety-critical systems. For example, recent work on a subset of Java for safety-critical systems (SCJ) is based on the programming model being defined in *SCJ-Circus* [6, 42, 7, 8], which is an extension to *Circus* whose semantics is defined using UTP. Together with formal models of the SCJ virtual machine, this allows the full semantics of an SCJ application to be defined [9].

SCJ is conservative in order to comply with guidelines for certification, such as DO-178C [25]; however, within the SCJ development team, there is the recognition that high-integrity software is generally becoming progressively more complex. To this end, they define different compliance levels. The most expressive programming model is supported at Level 2 compliance, and the SCJ team accept that certification of Level 2 applications requires significantly more effort and evidence than at Level 0 or Level 1 compliance. Even the Level 2 programming model is unable to exploit fully the power of the Java programming language due to the concerns over the ability to produce convincing certification evidence for programs that support dynamic class-loading, potentially across a network. It is also anticipated that some certification authorities may limit the use of other Java features to constrain the amount of dynamic dispatching and delegation that can occur in object-oriented programming languages (although the recent work in DO-178C shows that such techniques are becoming more accepted [25]).

The examples in this section illustrate some of the more dynamic behaviour that programs can exhibit, for which concise and intuitive formal models are required. The extension to CSP proposed in this paper provides semantics for techniques such as dynamic class-loading and the full use of dynamic dispatching and delegation. This can then be used in supporting evidence to allow certification of more complex systems to be considered in the future.

Example 1 (Frequent Flyer). Meyer gives an example of dynamic binding in Eiffel [17]: a person who is in a frequent flyer programme connects to a server with their membership number; they receive in reply a connection to another server according to their membership level: *Blue*, *Silver*, or *Gold*, and connections are made over the corresponding mobile channels *blue*, *silver*, and *gold*.

$$\begin{aligned}
FFConnect = {} & connect?p : MemberNo \rightarrow \\
& \quad \mathbf{if}\ p \in Blue\ \mathbf{then}\ service!blue \rightarrow SKIP \\
& \quad\quad \mathbf{else\ if}\ p \in Silver\ \mathbf{then}\ service!silver \rightarrow SKIP \\
& \quad\quad\quad \mathbf{else\ if}\ p \in Gold\ \mathbf{then}\ service!gold \rightarrow SKIP \\
& \quad\quad\quad\quad \mathbf{else}\ STOP
\end{aligned}$$

The process *FFConnect* serves a one-shot transaction. It waits for a membership number p input on the *connect* channel; it then analyses the value of p, and returns an appropriate channel name on the *service* channel. The code is a specification of the implementation in Eiffel that uses dynamic binding. □

Example 2 (Airline Check-in). An airline check-in system behaves as follows. The system consists of a collection of passengers, a clerk who assigns passengers to check-in desks, and the employees at the desks themselves. The behaviour of a passenger who wants to travel to a particular destination is as follows:

$$Passenger(dest) = \mathbf{new}\ p \bullet checkin!(p, dest) \rightarrow p?bc \rightarrow P(bc)$$

The passenger generates a fresh channel p for the visit to a desk, and then communicates that channel and the destination over the *checkin* channel to the clerk. The passenger then waits to receive a boarding card over channel p. The clerk receives the channel name and destination from a passenger and then waits for a desk to become free, which is signalled on the *next* channel with a channel name *cd*. The passenger then goes and does something else ($P(bc)$).

$$Clerk = checkin?(p, d) \rightarrow next?cd \rightarrow cd!(p, d) \rightarrow Clerk$$

The clerk then uses *cd* to inform the desk about the next passenger and their destination. $Desk(i)$ describes the behaviour of an airline representative. The representative generates the fresh channel name *cd* and sends it over the *next* channel for use by the clerk. The representative then waits to receive a communication on *cd* that tells them of the next passenger and their destination. The transaction is finalised by a reply on the passenger's channel p giving details of the boarding card $bcard(d)$.

$$Desk(i) = \mathbf{new}\, cd \bullet next!cd \rightarrow cd?(p, d) \rightarrow p!bcard(d) \rightarrow Desk(i)$$

The system is then given by

$$CheckIn = (\mid\mid\mid i : Desks \bullet Desk(i)) \parallel Clerk \parallel (\mid\mid\mid d : Today \bullet Passenger(d))$$

Today's destinations is a bag, with one destination for each passenger. □

In contrast to the previous example, this system does not involve dynamic binding as in OO languages, but instead a kind of dynamic binding of resources.

In occam-π, processes exchange the ends of channels [36]; as we see below, our theory is more powerful than this and involves mobile events. In spite of this, it is useful to describe a process's use of a channel in terms of the read-end or the write-end, and this usage can be checked syntactically. The following example uses Mobile CSP to model a simple two-place buffer using mobile channel ends. Of course, the obvious implementation would involve a single process using a linked-list data structure programmed using pointers. This may not be appropriate in a system with distributed memory, where a pointer in one memory space would have to address memory in another space.

Example 3 (Two-place Buffer). A user wants to read and write to a two-place buffer, and to do this, the user holds the input end of the *write* channel and the output end of the *read* channel. The buffer is made of two parallel processes connected by two channels, *chw* and *chr*. Between them, the two buffer processes hold the output end of the *write* channel and the input end of the *read* channel, and they swap ownership between themselves on the *chw* and *chr* channels, respectively. The state-transition for the buffer is pictured in Fig. 1, where the starting state has the left-hand process holding the buffer's ends of the *write* and *read* channels. The behaviour is:

$$\begin{aligned}
D_0(w, r) &= w?x \rightarrow chw!w \rightarrow D_1(x, r)\\
D_1(x, r) &= r!x \rightarrow chr!r \rightarrow D_2 \,\Box\, chw? \rightarrow D_2(x, w, r)\\
D_2 &= chw?w \rightarrow D_4(w)\\
D_3(x, w, r) &= r!x \rightarrow chr!r \rightarrow D_4(w)\\
D_4(w) &= chr?r \rightarrow D_0(w, r) \,\Box\, w?x \rightarrow D_5(x, w, r)\\
D_5(x) &= chr?r \rightarrow D_1(x, r)
\end{aligned}$$

The buffer has some invariant properties: where the buffer contains two elements or none (even parity), both ends reside in the same half of the buffer; where there is just a single element, the two channel ends reside in different halves, with the *read* end in the element's half. □

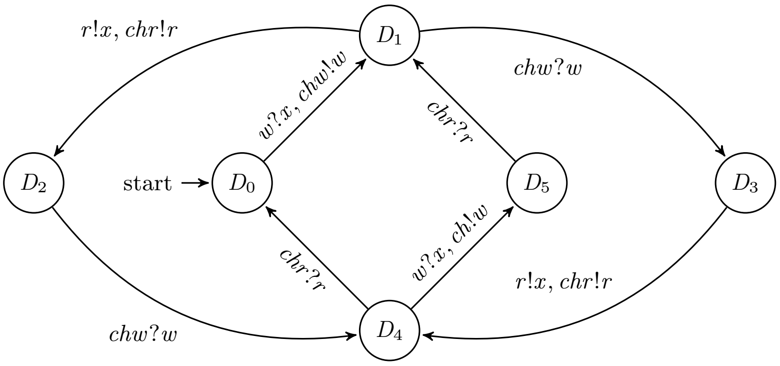


Fig. 1. STD for one-place buffer with mobile channel ends.

Example 4 (Ring Buffer). We can generalise Ex. 3 to an n-place ring buffer. Each cell in the buffer behaves as follows:

$$\begin{aligned} \mathit{Cell}(i) = {} & \mathit{ring}.i?(c?) \to c?x \to \mathit{ring}.(i+1 \;\mathsf{mod}\; n)!(c?) \to \\ & \mathit{ring}.i?(d!) \to d!x \to \mathit{ring}.(i+1 \;\mathsf{mod}\; n)!(d!) \to \mathit{Cell}(i) \end{aligned}$$

The cell starts by receiving the input end of a channel $c?$ over the channel $\mathit{ring}.i$; it then uses channel c to input a value x, which it buffers. The cell passes the channel end $c?$ on to the next cell in the ring; it does this by using the channel $\mathit{ring}.(i+1 \,\mathsf{mod}\, n)$. The cell waits for the output end of another channel $d!$, which it receives again on channel $\mathit{ring}.i$. It then outputs the value x, which it has been buffering, on channel d, before passing the channel end $d!$ to the next cell in the ring using $\mathit{ring}.(i+1 \;\mathsf{mod}\; n)$. Every cell is identical, starting with receiving the input end of a channel from its neighbour; so how does the buffer start being useful? The answer is to use one of the cells (it might as well be $\mathit{Cell}(0)$) in process $S(\mathit{in}?, \mathit{out}!)$. This process waits for the first input on the *in* channel, then passes the channel on to $\mathit{Cell}(1)$; it then waits to output its buffered value, passing the output channel on to $\mathit{Cell}(1)$; it then behaves like $\mathit{Cell}(0)$. In this way, the channel ends get into the ring.

$$S(\mathit{in}?, \mathit{out}!) = \mathit{in}?x \to \mathit{ring}.1!(\mathit{in}?) \to \mathit{out}!x \to \mathit{ring}.1!(\mathit{out}!) \to \mathit{Cell}(0)$$

The ring is then constructed from the cells, treating $\mathit{Cell}(0)$ to its initialisation:

$$\mathit{CellBuffer}(n) = S(\mathit{in}?, \mathit{out}?) \parallel \;\big\Vert\, i : 1 \,..\, n-1 \bullet \mathit{Cell}(i)$$

Process D in the previous example is simply a special case of this definition. □

Example 5 (Sieve of Eratosthenes). Primes models the Sieve of Eratosthenes and generates prime numbers on the channel c; it is composed initially of just two processes, one that generates natural numbers, and one that sifts them to

$\mathcal{A}$	event alphabet	physical and logical capabilities
$ok, ok' : \mathbb{B}$	stability	freedom from divergence
$wait, wait' : \mathbb{B}$	quiescence	waiting for interaction
$tr, tr' : \mathcal{A}^*$	trace	history of interaction
$ref, ref' : \mathbb{P}\,\mathcal{A}$	refusals set	events refused during wait
v, v'	program variables	imperative state

Table 2. Alphabet for CSP processes.

remove composite numbers:

$$Primes(c) = \mathbf{new}\ d \bullet Nats(2, d) \parallel Sift(d, c)$$
$$Nats(n, d) = d!n \to Nats(n + 1, d)$$

The process *Sift* spawns a series of filters, each removing composites:

$$Sift(in, out) = \mathbf{new}\ d \bullet in?p \to out!p \to Filter(p, in, d) \parallel Sift(d, out)$$
$$Filter(p, in, out) = \mu\, X \bullet in?x \to (out!x \to X \lhd x \mathsf{\ mod\ } p \neq 0 \rhd X)$$

The mobile channels are used to build an unbounded process structure: we can start *Primes* as a prime number server in some larger system, knowing that it will run indefinitely (well, until the underlying resources required for channels are exhausted). The obvious alternative implementation is to declare a sufficiently large number of channels in advance, and then to use these one by one. The difference between these approaches is similar to lazy versus eager evaluation in functional programming, and the advantages are the same. □

4 Semantic Domain

Hoare & He give the semantic domain for CSP in UTP [12, Chap. 8] (see also [40, 4] for tutorial introductions). In their semantics, each process is represented by an alphabetised predicate arranged in a lattice ordered by refinement, which is defined as universally closed inverse implication. The alphabet describes the observations that can be made of processes, and these are summarised in Tab. 2. Each predicate in the lattice is actually a relation between a before-state (ok, $wait$, tr, ref, and v) and an after-state (ok', $wait'$, tr', ref', and v'); the alphabet $\mathcal{A}$ is a constant. Membership of the lattice is defined by the fixed-points of five functions representing healthiness conditions, and these are summarised in Tab. 3. The predicates in this lattice can also be expressed as ***R***-healthy precondition-postcondition pairs: "reactive designs" [21] (where ***R*** is the composition of ***R1*** to ***R3***). The precondition describes the conditions under which the process does not diverge, while the postcondition describes its failures.

Now we can define the semantic domain for mobile CSP. The obvious idea is to make the alphabet a dynamic variable; however, a moment's thought shows this is inadequate because of compositionality: we need the dynamic alphabet's

never undo	***R1***	$P \;=\; P \wedge (tr \le tr')$
ignore history	***R2***	$P(tr, tr') \;=\; \bigsqcap s \bullet P(s, s \frown (tr' - tr))$
wait!	***R3***	$P \;=\; (\mathit{II}_{\mathbf{R}} \lhd wait \rhd P)$
diverge	***CSP1***	$(\neg\, ok \wedge tr \le tr') \vee P$
ok'-monotonicity	***CSP2***	$P \;;\; J$

$$\mathit{II}_{\mathbf{R}} \;\widehat{=}\; (\neg\, ok \wedge tr \le tr') \vee (ok' \wedge (tr' = tr) \wedge \cdots \wedge (v' = v))$$
$$J \;\widehat{=}\; (ok \Rightarrow ok') \wedge (wait' = wait) \wedge (tr' = tr) \wedge (refr' = ref) \wedge (v' = v)$$

Table 3. Healthiness conditions for CSP processes.

history when we come to compose the traces of two parallel processes. For example, consider the process that executes an event a and then enters a state with alphabet $\{a, b\}$. What can we say about the alphabet *before* the a event? It certainly must include a itself, but what about b? We need to know the answer in order to know whether the process has right of veto over b in any composition. A better strategy is to record a process's alphabet before and after each event.

Definition 1 (Dynamic Alphabetised Traces (DATs)). *A DAT is non-empty and alternates alphabets and events, starting with an alphabet:*

$$DAT^{\epsilon} \;\widehat{=}\; \{\, s \mid \#s \in Odd \wedge odds(s) \in (\mathbb{P}\,\Sigma)^* \wedge evens(s) \in (\Sigma \cup \{\epsilon\})^* \,\}$$

The silent event ϵ is used below to define the **new** *and* **dispose** *commands that manipulate a process's alphabet.* □

Example 6 (Dynamic Alphabetised Traces). The following are all valid DATs

$$\langle\{b\}\rangle \qquad \langle\{b\}, b, \{a, b\}\rangle \qquad \langle\{b\}, b, \{a, b\}, a, \{a, b\}\rangle$$

(Here, Odd is the set of odd integers; $odd(s)$ is the sequence of s's odd-indexed elements; and $even(s)$ is the sequence of s's even-indexed elements. □

We can now express some simple properties over DAT traces:

- start, owning c: $\langle\{c\}\rangle$
- acquire event c: $\langle\cdots, \{a, b\}, a, \{a, b, c\}, \cdots\rangle$
- release event c: $\langle\cdots, \{a, b, c\}, a, \{a, b\}, \cdots\rangle$

Mobile processes satisfy two healthiness conditions on their DAT, tr.

Definition 2 (Ownership). *A mobile process can engage in an event only if it has already acquired it, but not released it.*

$$\textbf{\textit{M1}} \quad P \;=\; P \wedge (\,\forall\, s : \mathbb{P}\,\Sigma;\; e : \Sigma \bullet \langle s, e\rangle \in \operatorname{ran} tr' \Rightarrow e \in s\,)$$

This is defined using a monotonic idempotent healthiness condition. □

Definition 3 (Refusalship). *A mobile process can refuse only those events it has acquired.*

$$\boldsymbol{M2} \quad P \;=\; P \wedge \mathit{ref}' \subseteq \mathit{last}\, tr'$$

Again, this is enforced by a monotonic idempotent healthiness condition. □

The two healthiness conditions commute. The reactive healthiness conditions must hold, but with $\boldsymbol{R2}$ for $(\mathit{Even} \lhd tr)$ and $(\mathit{Even} \lhd tr')$, (the operator $\lhd$ is domain restriction of a function). $\boldsymbol{R2}(P)$ ensures compositionality by insisting that P does not depend on particular values for tr. In the sequence $P \;;\; Q$, the final alphabet in the trace of P must match the initial alphabet of Q. Finally, concatenation between dynamic alphabet traces is a partial function:

$$\mathit{last}\; t = \mathit{head}\; u \Rightarrow t \curvearrowright u = t \frown (\mathit{tail}\; u)$$

The $\boldsymbol{M}$ healthiness conditions commute with the $\boldsymbol{R}$ healthiness conditions.

5 Semantics of Operators

In this section, space allows us to give the semantics for a few key constructs. We do this in the style of reactive designs [21], as described in Sect. 4. The definition of a UTP design with precondition P and postcondition Q is [12]:

$$P \vdash Q \;\widehat{=}\; (ok \wedge P \Rightarrow ok' \wedge Q)$$

That is, if the design is started in a stable state (ok) and the precondition is true (P), then it must reach a stable state (ok') and when it does so, the postcondition will be true (Q). This is a statement of total correctness.

5.1 Event Prefixes (Value + Channel)

The semantics of an event-prefixed process, $a \rightarrow \mathit{SKIP}$, depends on whether the event a is the communication of a channel name. If it is not, then the UTP semantics is similar to that in standard CSP [12]:

Definition 4 (Event Prefix (Value)).

$$\boldsymbol{M} \circ \mathbf{R}(a \in \mathit{last}\; tr \vdash tr' = tr \wedge a \notin \mathit{ref}' \lhd \mathit{wait}' \rhd tr' = tr \frown \langle a, \mathit{last}\; tr \rangle)$$

The precondition requires that a is in the current alphabet, the last entry of the trace preceding execution of the process $(\mathit{last}\; tr)$; the precondition in standard CSP is simply **true**. Since tr is a DAT, it ends with an alphabet, so $\mathit{last}\; tr$ is well defined. The postcondition has a small difference too: if the process terminates, then $\langle a, \mathit{last}\; tr \rangle$ is appended to the trace ($\curvearrowright$ is not needed here); the current alphabet is unchanged by this kind of event. If the event a is not in the current alphabet, then the design aborts and the process diverges: this is a channel fault.

Now consider $c?n \rightarrow \mathit{SKIP}$, which inputs channel name n over c.

Definition 5 (Event Prefix (Channel Name)).

$$\boldsymbol{M} \circ \boldsymbol{R}\left(\begin{array}{l} c.n \in \mathit{last}\ \mathit{tr} \\ \vdash \mathit{tr}' = \mathit{tr} \wedge c.n \notin \mathit{ref}' \lhd \mathit{wait}' \rhd \mathit{tr}' = \mathit{tr} \frown \langle c.n, (\mathit{last}\ \mathit{tr}) \cup \{\!|n|\!\} \rangle \end{array}\right)$$

The difference is the expansion of the alphabet with $\{\!|n|\!\}$, which is the set of all events communicable over n. Outputting a name is complementary.

5.2 New and Dispose

In UTP, a block-structured declaration **var** $x \bullet P$ is semantically equivalent to the predicate **var** x ; P ; **end** x, where the beginning and end of the scope of x are treated separately [12, Chap. 2]. We adopt a similar approach to the block-structured allocation of fresh channels **new** $c \bullet P$, and deal separately with the allocation and disposal of a channel: **new** c ; P ; **dispose** c.

Definition 6 (New Channel). *For fresh c,*

$$\mathbf{new}\ c\ ;\ P \mathrel{\widehat{=}} \boldsymbol{M} \circ \boldsymbol{R}(\,\mathbf{true} \vdash \neg\, \mathit{wait}' \wedge \mathit{tr}' = \mathit{tr} \frown \langle \epsilon, (\mathit{last}\ \mathit{tr}) \cup \{\!|c|\!\} \rangle\,)$$

Definition 7 (Dispose Channel).

$$\mathbf{dispose}\ c\ ;\ P \mathrel{\widehat{=}} \boldsymbol{M} \circ \boldsymbol{R}(\,\mathbf{true} \vdash \neg\, \mathit{wait} \wedge \mathit{tr}' = \mathit{tr} \frown \langle \epsilon, (\mathit{head}\ \mathit{tr}') \setminus \{\!|c|\!\} \rangle\,)$$

Example 7 (Channel Allocation). Consider the $\mathit{Desk}(i)$ process in Ex. 2:

$$\mathbf{new}\ \mathit{cd} \bullet \mathit{next}!\mathit{cd} \rightarrow \mathit{cd}?(p, d) \rightarrow p!\mathit{bcard}(d) \rightarrow \mathit{Desk}(i)$$

The process must initially own the *next* channel's events: $\{\!|\mathit{next}|\!\}$; these events are all channel name communications. Here is an example trace:

$$\begin{array}{l} \langle\ \{\!|\mathit{next}|\!\}, \\ \qquad \epsilon, \{\!|\mathit{next}|\!\} \cup \{\!|\mathit{cd}|\!\}, \\ \qquad\quad \mathit{next}.\mathit{cd}, \{\!|\mathit{next}|\!\} \cup \{\!|\mathit{cd}|\!\}, \\ \qquad\qquad \mathit{cd}.(p, d), \{\!|\mathit{next}|\!\} \cup \{\!|\mathit{cd}|\!\} \cup \{\!|p|\!\}, \\ \qquad\qquad\quad p.\mathit{bcard}(d), \{\!|\mathit{next}|\!\} \cup \{\!|\mathit{cd}|\!\} \cup \{\!|p|\!\}, \\ \qquad\qquad\qquad \epsilon, \{\!|\mathit{next}|\!\} \cup \{\!|p|\!\}\ \rangle \end{array}$$

In this trace, *cd* is a fresh channel name; p is bound to a channel name input as part of the pair on the *cd* channel. Notice how we automatically dispose of *cd* at the end of the process; however, there is no automatic disposal of the channel denoted by p. A better definition for the process would tidy this up:

$$\mathbf{new}\ \mathit{cd} \bullet \mathit{next}!\mathit{cd} \rightarrow \mathit{cd}?(p, d) \rightarrow p!\mathit{bcard}(d) \rightarrow \mathbf{dispose}\ p;\ \mathit{Desk}(i)$$

Here, the events of whichever channel is denoted by p are removed from the alphabet when the scope of the channel variable p ends. □

5.3 Parallel Composition

In UTP, parallel composition uses the parallel-by-merge semantic pattern taken from Hoare & He's UTP semantics for ACP, CCS, and CSP [12]: two processes have their overlapping alphabets separated by renaming; they are then run in parallel producing two states, which are then merged to give the meaning of the composition. We need to specify only the merge for DAT traces.

Definition 8 (Parallel Merge). *Define a "catset" operator that concatenates its left-hand sequence operand with every sequence in its right-hand set operand:*

$$s \mathbin{\overset{\frown}{*}} T \ \mathrel{\widehat=}\ \{\, u : T \bullet s \frown u \,\}$$

We use this operator to define the parallel composition of two DAT traces:

$$\begin{aligned}
&\langle s\rangle \frown xs \parallel \langle t\rangle \frown ys \ \mathrel{\widehat=}\ \langle s \cup t\rangle \mathbin{\overset{\frown}{*}} N(s, xs, t, ys)\\
&N(s, \langle\rangle, t, ys) = \{ys\}\\
&N(s, xs, t, \langle\rangle) = \{xs\}\\
&N(s, \langle x\rangle \frown xs, t, \langle y\rangle \frown ys) =\\
&\qquad \textbf{if } x = y \neq \epsilon \textbf{ then } \langle x\rangle \mathbin{\overset{\frown}{*}} (xs \parallel ys)\\
&\qquad \textbf{else } (\,\textbf{if } x \notin t \textbf{ then } \langle x\rangle \mathbin{\overset{\frown}{*}} (xs \parallel \langle t, y\rangle \frown ys)\,)\\
&\qquad\qquad \cup (\,\textbf{if } y \notin s \textbf{ then } \langle y\rangle \mathbin{\overset{\frown}{*}} (\langle s, x\rangle \frown xs \parallel ys)\,)
\end{aligned}$$

Silent events occur independently: if two parallel processes each allocate a new channel, then there is no synchronisation of the two **new** *commands. It is easily shown by induction that the merge operator is closed on DAT traces.* □

Example 8 (Parallel Merge). Consider the two Mobile CSP processes: $P = a \rightarrow SKIP$ and $Q = get.a \rightarrow a \rightarrow SKIP$.

P's behaviour includes the following trace: $\langle\{a\}, a, \{a\}\rangle$; Q's behaviour includes $\langle\{get.a\}, get.a, \{get.a, a\}, a, \{get.a, a\}\rangle$. The parallel composition of the two traces describes two behaviours: the first has P executing a *before* Q gets hold of a ($\langle\{get.a, a\}, a, \{get.a, a\}, get.a, \{get.a, a\}\rangle$; the second has P executing a *afterwards* Q ($\langle\{get.a, a\}, get.a, \{get.a, a\}, a, \{get.a, a\}\rangle$. In the first trace, P executes a independently; in the second trace P and Q synchronise on a. □

6 Implementation

The ring buffer described in Ex. 4 can be implemented in pure CSP:

$$\begin{aligned}
Imp(i) = {}& ring.i?c \rightarrow chan.c?x \rightarrow ring.((i+1) \mathbin{\mathsf{mod}} n)!c \rightarrow\\
& \quad ring.i?d \rightarrow chan.d!x \rightarrow ring.((i+1) \mathbin{\mathsf{mod}} n)!d \rightarrow Imp(i)
\end{aligned}$$

Here, we simulate mobile channels by passing around tokens so that the end of a channel can be used only by the process that holds the token for that channel end, whilst those processes without a token do not block. We use a token for the

```
package mobileCode;
public interface ServerInterface {
   public void useService(String parameters);
}
```

Fig. 2. `ServerInterface.java`

```
public enum MembershipLevel {Blue, Silver, Gold}
public class Broker {
   // directory of servers implemented via a Java Map
   public ServerInterface lookUpService(MembershipLevel l) {
     ServerInterface server;
     // lookup server
     return server;
  }
   public synchronized void register(ServiceProvider serverThread,
                                  MembershipLevel level) {
     // save details in map
  }
}
```

Fig. 3. `MembershipLevel.java`

relevant channel name to index an array of channels *chan*; interleaving is needed to share the channel ends (see [11] on shared resources).

Examples 1 and 2 use the *broker* architectural pattern [2], suitable for distributed systems where clients invoke remote services, but are unconcerned with the details of remote communication. In systems engineering, there are many practical reasons to adopt a distributed architecture. The system may need to take advantage of multiple processors or a cluster of low-cost computers. Certain software may be available only on specific computers, or provided by third parties and available on the cloud. The broker pattern can hide many of these implementation issues by encapsulating required services into a separate layer.

Example 9 (Broker Pattern). In SCJ, the broker pattern is an application that requires dynamic dispatching through interfaces, as illustrated in Fig. 2. The broker itself simply maintains a directory of service providers. In Ex. 1, there are three service providers for *Blue*, *Silver* and *Gold* membership levels, as shown in Fig. 3. These register with the broker via a synchronised method. To simplify the example, assume that the system runs on a multiprocessor server, and the service providers have their own resources allocated. The application is encapsulated in an SCJ mission (subsystem) and the service providers are managed threads that implement the service interface, as shown in Fig. 4. The threads are Java daemon threads, terminating automatically. Finally, the clients are also managed threads

```
public class ServiceProvider extends ManagedThread implements ServerInterface {
   @Override
   public void useService(String s) {
     // add to queue of requests
     // wait until search has been performed
     return;
   }
   @Override
   public void run() {
     broker.register(this, level);
     while (true) {
       // perform services while needed
     }
   }
   public ServiceProvider(Broker b, MembershipLevel l) {
     super();
     this.broker = b;
     this.level = l;
     this.setDaemon(true);
   }
   private Broker broker;
   private MembershipLevel level;
}
```

Fig. 4. `ServiceProvider.java`

that are assigned a particular membership level when created (Fig. 5). To analyse an program with the broker pattern requires all possible classes implementing `ServiceInterface` to provide equivalent functionality. For a large system, where the same broker is used to provide the interface between many service providers and clients, this may be difficult to guarantee and to provide evidence that each service provider is being used in the correct context. □

The broker acts as a messenger: locating an appropriate server; forwarding requests to that server, possibly marshalling data; and transmitting results to the client, possibly demarshalling data. Clients are applications that access servers, and they call the remote service by forwarding requests to the broker and receiving responses or exceptions in reply. Widely used broker patterns include OMG's CORBA standard, Microsoft's Active X, and the World-Wide Web, where browsers act as brokers and servers act as service providers.

7 Related Work

Our traces component is essentially the same as that of Hoare & O'Hearn [13]; they explore the unification of CSL (concurrent separation logic) and CSP by

```
public class Client extends ManagedThread {
  private Broker broker;
  private MembershipLevel level;
  public Client(Broker b, MembershipLevel l) {
    broker = b; level = l;
  }
  @Override
  public void run() {
    // code for client
    ServerInterface myProvider = broker.lookUpService(level);
    myProvider.useService(params);
  }
  private String params;
}
```

Fig. 5. `Client.java`

adding temporal separation to CSL and mobile channels to CSP, restricting to the traces model, excluding nondeterminism, deadlock, and livelock. Their interest lies in point-to-point communication, using ideas from separation logic to reason about exclusive use of channel ends, used as in occam-π [36]. Processes that send a channel end automatically relinquish ownership. This differs from our work, where channels may have many ends and ownership must be handled explicitly. Actually, the structure of a trace is slightly different: instead of simple events, they allow sets of events, giving a semantics for true concurrency and having no need for ϵ. Their work, with its deliberate restrictions, leads to a very simple notion of event composition using point-wise disjoint union.

Roscoe discusses a version of CSP with mobility [24, Sect. 20.3], so that all processes that do not presently have the right to use a particular action always accept the event and never change their state when the event occurs. This is the same as our route to implementation in Sect. 6. He has shown how a full semantics of the π-calculus can be given in CSP [23]: for each π-calculus agent, there is a CSP process that models it accurately.

In Mobile CSP$\|$B [34], controllers can work with different machines during execution. Controllers can exchange machines between each other by exchanging machine references and manage concurrent state updates. This work goes further than the current paper and the references cited in this section by dealing with the fusion of concurrency, communication, state, and channel mobility.

8 Conclusions and Future Work

This paper has explored a semantics for mobile CSP for specifying and verifying safety-critical code. It permits the use of dynamic programming features, such as the use of the broker architectural and programming pattern, in a controlled fashion so that evidence for assurance can be collected and relied upon. The work started as a contribution to occam-π; more recently, it has contributed to Safety-Critical Java and on reasoning about systems of systems and cyber-physical systems. The work is still at an early stage, and there are plenty of directions for future work. We will give a complete account of the operators of mobile CSP and extend this to *MobileCircus*. We need to prove closure of all these operators with respect to $\boldsymbol{R}$ and $\boldsymbol{M}$, and perhaps devise additional healthiness conditions. We have hinted at the possibility of translating mobile CSP into plain CSP; we need to record a Galois connection between the two and use it as the basis of a translator. Finally, since our intention is to verify system architectures and programs, we will devise a Hoare logic for Mobile CSP and prove it sound.

9 Acknowledgements

This work has been supported by the EPSRC hiJaC and the EU H2020 INTO-CPS projects. Thanks to Philippa Gardiner for an application of the π-calculus that led to Ex. 2; to Peter Welch for discussions more than a decade ago on mobile channels in occam-π that led to the semantic model in this paper; and to three anonymous referees for prompting clarifications in the paper.

References

1. John Bjørndalen et al. PyCSP — CSP for Python. In Alistair McEwan et al., editors, *CPA*, pages 229–248, 2007.
2. Frank Buschmann. *Pattern-Oriented Software Architecture*. Wiley, 1996.
3. Andrew Butterfield et al. *Slotted-Circus*. In Jim Davies et al., editors, *IFM*, volume 4591 of *LNCS*, pages 75–97. Springer, 2007.
4. Ana Cavalcanti et al. A tutorial introduction to CSP in *Unifying Theories of Programming*. In Ana Cavalcanti et al., editors, *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2004.
5. Ana Cavalcanti et al. Unifying classes and processes. *SoSyM*, 4(3):277–296, 2005.
6. Ana Cavalcanti et al. Safety-critical Java in *Circus*. In Andy Wellings et al., editors, *JTRES*, pages 20–29, 2011.
7. Ana Cavalcanti et al. The Safety-critical Java memory model: A formal account. In M. J. Butler et al., editors, *FM 2011*, volume 6664 of *LNCS*, pages 246–261. Springer, 2011.
8. Ana Cavalcanti et al. The Safety-critical Java memory model formalised. *Formal Asp. Comput.*, 25(1):37–57, 2013.
9. Ana Cavalcanti et al. Safety-critical Java programs from *Circus* models. *Real-Time Systems*, 49(5):614–667, 2013.
10. Clemens Fischer. Combining Object-Z and CSP. In Adam Wolisz et al., editors, *Formale Beschreibungstechniken für verteilte Systeme*, volume 315 of *GMD-Studien*, pages 119–128, 1997.
11. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
12. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall, 1998.
13. Tony Hoare and Peter W. O'Hearn. Separation logic semantics for communicating processes. *ENTCS*, 212:3–25, 2008.

14. ISO. *ISO/IEC 8652:2012 Information technology — Prog. languages — Ada*, 2012.
15. Gabor Karsai. Unification or integration? The challenge of semantics in heterogeneous modeling languages. In Benoît Combemale et al., editors, *The Globalization of Modeling Languages*, pages 2–6, 2014.
16. Z. Liu et al. rCOS: Refinement of component and object systems. In F. de Boer et al., editors, *FMCO*, volume 3657 of *LNCS*, pages 183–221. Springer, 2004.
17. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.
18. Robin Milner et al. A calculus of mobile processes. *Inf. Comput.*, 100:41–77, 1992.
19. FDR3 model checker. `www.cs.ox.ac.uk/projects/fdr/`.
20. Martin Odersky. Programming in Scala, 2008. Mountain View, California.
21. Marcel Oliveira et al. A denotational semantics for *Circus*. *Electr. Notes Theor. Comput. Sci.*, 187:107–123, 2007.
22. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.
23. A .W .Roscoe. CSP is expressive enough for π. In C. B. Jones, A. W. Roscoe, and K. R. Wood, editors, *Reflections on the Work of C. A. R. Hoare.* Springer, 2010.
24. A. W. Roscoe. *Understanding Concurrent Systems.* Springer, 2010.
25. RTCA. Object-oriented technology and related techniques supplement to DO-178C [ED-12C] and DO-178A. Technical Report DO-332/ED-217, [ED-109A], 2011.
26. Steve Schneider and Helen Treharne. Communicating B machines. In Didier Bert et al., editors, *ZB 2002*, volume 2272 of *LNCS*, pages 416–435. Springer, 2002.
27. Adnan Sherif et al. A process algebraic framework for specification and validation of real-time systems. *Formal Asp. Comput.*, 22(2):153–191, 2010.
28. Neil Smyth. Communicating Sequential Processes in Ptolemy II. Tech. Memo. UCB/ERL M98/70, Electronics Research Laboratory, Berkeley, December 1998.
29. Jun Sun et al. Model checking CSP revisited: Introducing a process analysis toolkit. In Tiziana Margaria et al., editors, *ISoLA*, pages 307–322. Springer, 2008.
30. Xinbei Tang et al. Towards mobile processes in unifying theories. In *SEFM*, pages 44–53, 2004.
31. Xinbei Tang et al. Travelling processes. In *MPC*, volume 3125 of *LNCS*, pages 381–399. Springer, 2004.
32. inmos. *occam Programming Manual.* Prentice Hall, 1984.
33. H. Treharne and S. Schneider. How to drive a B machine. In Jonathan P. Bowen et al., editors, *ZB 2000*, volume 1878 of *LNCS*, pages 188–208. Springer, 2000.
34. Beeta Vajar et al. Mobile CSP||B. *ECEASST*, 23, 2009.
35. P. H. Welch et al. The JCSP (CSP for Java) Home Page, 1999.
36. Peter H. Welch et al. Mobile barriers for occam-π: Semantics, implementation and application. In Jan F. Broenink et al., editors, *CPA*, pages 289–316, 2005.
37. Jim Woodcock. Engineering UToPiA—Formal semantics for CML. In *FM 2014*, volume 8442 of *LNCS*, pages 22–41. Springer, 2014.
38. Jim Woodcock et al. A concurrent language for refinement. In Andrew Butterfield et al., editors, *5th Irish Workshop on Formal Methods*, 2001.
39. Jim Woodcock et al. The semantics of *Circus*. In *ZB 2002*, volume 2272 of *LNCS*, pages 184–203. Springer, 2002.
40. Jim Woodcock et al. A tutorial introduction to designs in Unifying Theories of Programming. In Eerke A. Boiten et al., editors, *Integrated Formal Methods, IFM 2004*, volume 2999 of *LNCS*, pages 40–66. Springer, 2004.
41. Jim Woodcock et al. Features of CML: A formal modelling language for systems of systems. In *ICOSE*, pages 445–450, 2012.
42. Frank Zeyda et al. The Safety-critical Java mission model: A formal account. In Shengchao Qin et al., editors, *ICFEM 2011*, volume 6991 of *LNCS*, pages 49–65. Springer, 2011.

Author Index