



REYKJAVÍK UNIVERSITY  
HÁSKÓLINN Í REYKJAVÍK

*Proceedings of the 27th Nordic Workshop  
on Programming Theory (NWPT 2015)*

Luca Aceto, Ignacio Fábregas, Álvaro García-Perez, Anna Ingólfssdóttir

RUTR-SCS16001, February 2016  
School of Computer Science

Reykjavík University - School of Science and Engineering

**Technical Report**

ISSN 1670-5777





REYKJAVÍK UNIVERSITY  
HÁSKÓLINN Í REYKJAVÍK

## Proceedings of the 27th Nordic Workshop on Programming Theory (NWPT 2015)

Luca Aceto, Ignacio Fábregas, Álvaro García-Perez, Anna Ingólfssdóttir

School of Computer Science  
Technical Report RUTR-SCS16001 – February 2016

*(Útdráttur: næsta síða)*



HÁSKÓLINN Í REYKJAVÍK  
REYKJAVÍK UNIVERSITY

## Proceedings of the 27th Nordic Workshop on Programming Theory (NWPT 2015)

Luca Aceto, Ignacio Fábregas, Álvaro García-Perez, Anna Ingólfssdóttir

Tölvunarfræðisvið  
Tækniskýrsla RUTR-SCS16001 – Febrúar 2016

*(Abstract: previous page)*

# Contents

<b>Preface</b>	<b>1</b>
<b>Invited Talks</b>	<b>1</b>
Languages and Models for Collective Adaptive Systems . . . . .	2
Computing Reliably with Molecular Walkers . . . . .	3
Techniques and Tools for the Analysis of Timed Workflows . . . . .	4
<b>Papers presented</b>	<b>5</b>
Limitations of Non-Interference . . . . .	5
Towards Component-based Reuse for Event-B . . . . .	10
Improving Partial Order Reductions in Multithreaded Programs with Local First Search . . . . .	13
Modelling and analysis of normative contracts . . . . .	16
From Explicit to Implicit Dynamic Frames in Concurrent Reasoning for Java (Extended Abstract) . . . . .	19
A Formal Framework Supporting Unrestricted Software Changes in Object-Oriented Concurrent Systems . . . . .	23
An implementation in Agda of Sutner’s decision algorithms for injectivity and surjectivity of one-dimensional cellular automata . . . . .	27
Rule formats for bounded nondeterminism in nominal structural operational semantics . . . . .	30
Winning Cores in Parity Games . . . . .	35
Privacy in Evolving Social Networks . . . . .	39
Formally Verifying Exceptions for Low-level code with Separation Logic . . . . .	42
Specialized Strategies for Learning Integrated Circuits using Angluin L* and Rivest/Shapire Homing Inference . . . . .	45
Binary session types for psi-calculi . . . . .	48
Towards A Hybrid Approach to Software Verification (Extended Abstract) . . . . .	51
Compiling Protocol Narrations into Applied Pi Processes . . . . .	54
A generalization of termination conditions for partial model completion . . . . .	56
Flooding Detection in Concurrent Object Systems . . . . .	60
Business Process Conformance Checking Based on Event Structures . . . . .	63
Formal Verification using Parity Games . . . . .	66
Join inverse categories and reversible recursion . . . . .	69
Errors as Data Values as the Language Default . . . . .	72
Discounted Duration Calculus . . . . .	75
Algebraic Combinators for Data Dependencies and Their Applications . . . . .	78
Contract-based specification and verification of dataflow programs . . . . .	81
Tool Support for Component-Based Semantics . . . . .	84
A logical characterisation for input output conformance simulation iocos . . . . .	87
Using Typings as Types . . . . .	90
Towards user-friendly and efficient analysis with Alloy . . . . .	94
A Property Specification Language for Runtime Verification of Executable Models . . . . .	97
On endofunctors modelling higher-order behaviours . . . . .	100
Open memory transactions in Haskell . . . . .	104
Extending a Theorem Prover for Deductive Program Verification . . . . .	107
Session-Based Compositional Verification on Actor-based Concurrent Systems . . . . .	110
StaRVOOrS: Unifying Static and Runtime Verification of Java . . . . .	113
A formal model for direct-style asynchronous observables . . . . .	116
Probabilistic Floating-Time Transition System: A New Approach For State Space Reduction of Probabilistic Actors . . . . .	119
Towards Small-step Compilation Schemas for SOS . . . . .	121
Acyclic attribute evaluation in a dependently typed setting . . . . .	124
Implementation-based Refinements for Equivalence Class Testing . . . . .	127



## Preface

This volume includes the abstracts that were selected for presentation at the 27th Nordic Workshop on Programming Theory (NWPT 2015). The workshop was held at Reykjavik University in the period 21–23 October 2015 and was organized by us in cooperation with Dario Della Monica.

The event had 57 registered participants (50 of which came from abroad, giving yet another indication of the powerful lure of Iceland as a destination for scientific events), and several talks were also attended by some local faculty members and students who were not officially registered for the workshop. All sessions were well attended and had lively discussions, including the very last one.

The workshop was graced by three excellent invited talks by Rocco De Nicola, Marta Kwiatkowska and Jiri Srba, and the quality of the contributed presentations was consistently high. It was very pleasing to see many young researchers deliver clear, well prepared and well paced presentations.

The abstracts for the contributed presentations and the slides for nearly all the talks are available at <http://icetcs.ru.is/nwpt2015/programme.html>.

We thank the members of the PC for the workshop and all the participants for making NWPT 2015 a scientifically interesting and enjoyable event.

Luca Aceto, Ignacio Fábregas, Álvaro García-Perez, Anna Ingólfssdóttir

ICE-TCS, School of Computer Science, Reykjavik University.

# Languages and Models for Collective Adaptive Systems

Rocco De Nicola

IMT Lucca, Italy

## Abstract

Collective Adaptive Systems (CAS) are systems that consist of a large number of interacting components that dynamically adjust and combine their behaviour to achieve specific goals. We propose a set of programming abstractions that have been specifically designed to deal with CAS, and with their need to adapt to the changes of the working environment and to the evolving requirements. Based on these abstractions, we introduce SCEL (Software Component Ensemble Language), a kernel language whose solid semantic foundations lay also the basis for formal reasoning on CAS. One of the key feature of SCEL is the so called attribute-based communication, an alternative to broadcast and to binary communication. Building on this, we introduce also a basic process calculus, named AbC, whose primary primitive for interaction is exactly attribute-based communication. An AbC system consists of a set of parallel components each of which is equipped with a set of attributes. Communication takes place in an implicit multi-cast fashion, and interactions among components are dynamically established by taking into account connections as determined by predicates over the attributes exposed by components. Expressiveness and effectiveness of AbC are demonstrated both in terms of the ability to model scenarios featuring collaboration, reconfiguration, and adaptation and of the possibility of encoding a process calculus for broadcasting channel-based communication and other communication paradigms. Behavioural equivalences for AbC are introduced for establishing formal relationships between different descriptions of the same system.



# Computing Reliably with Molecular Walkers

Marta Kwiatkowska

University of Oxford, UK

## Abstract

DNA computing is emerging as a versatile technology that promises a vast range of applications, including biosensing, drug delivery and synthetic biology. DNA logic circuits can be achieved in solution using strand displacement reactions, or by decision-making molecular robots-so called 'walkers'-that traverse tracks placed on DNA 'origami' tiles.

Similarly to conventional silicon technologies, ensuring fault-free DNA circuit designs is challenging, with the difficulty compounded by the inherent unreliability of the DNA technology and lack of scientific understanding. This lecture will give an overview of computational models that capture DNA walker computation and demonstrate the role of quantitative verification and synthesis in ensuring the reliability of such systems. Future research challenges will also be discussed.

# Techniques and Tools for the Analysis of Timed Workflows

Jiri Srba

Aalborg University, Denmark

## Abstract

Analysis of workflow processes with quantitative aspects like timing is of interest in numerous time-critical applications. In this talk, I will suggest a workflow model based on timed-arc Petri nets and study the foundational problems of soundness and strong (time-bounded) soundness. We will explore the decidability of these problems and show, among others, that soundness is decidable for monotonic workflow nets while reachability is undecidable. For general timed-arc workflow nets soundness and strong soundness become undecidable, though we can design efficient verification algorithms for the practically interesting subclass of bounded nets. Finally, I demonstrate the usability of the theory on a few case studies of a Brake System Control Unit used in aircraft certification, the MPEG2 encoding algorithm, a blood transfusion workflow and a home automation system for a family house. The implementation of the algorithms is freely available as a part of the model checker TAPAAL ([www.tapaal.net](http://www.tapaal.net)|<http://www.tapaal.net>).

# Limitations of Non-Interference

Flemming Nielson   Hanne Riis Nielson   Ximeng Li

DTU Compute, Technical University of Denmark, Denmark

{fnie,hrni,ximl}@dtu.dk

Submitted to NWPT 2015

## Abstract

We show that non-interference falls short of providing a convincing semantic characterisation of information flow policies for confidentiality and integrity and motivate an approach based on instrumented semantics.

**Introduction.** We have been working with Airbus on developing security policies for dealing with the challenges of communication between security domains subject to strict safety concerns, and in the course of this work we have uncovered a limitation of non-interference in establishing convincing semantic characterisations of the required security policies. In this paper we illustrate this limitation on an utterly simple example and discuss ways of providing alternate semantic characterisations more acceptable to our industrial partners.

**An illustrative example.** Let us consider a simple process  $D$  that takes inputs  $a$ ,  $b$ , and  $c$ , and produces outputs  $d1=a+b$  and  $d2=b*c$ .

```
1  process D
2  begin
3    input(a,b,c);
4    d1:=0; d2:=1;
5    d1:=d1+a; d2:=d2*b;
4    d1:=d1+b; d2:=d2*c;
7    output(d1,d2)
8  end
```

In the full development, the inputs would be received from other parallel processes and the outputs would be delivered to other parallel processes. Here we simply assume that the variables  $a$ ,  $b$ ,  $c$  belong to the processes  $A$ ,  $B$ ,  $C$ , respectively, and that the variables  $d1$  and  $d2$  both belong to the process  $D$ .

**Security policies.** Motivated by the Decentralized Label Model [3] the security policies of interest have two components. One component,  $R$ , tracks where the values of variables are allowed to flow and is useful for dealing with confidentiality; we shall say that it tracks the *readers* of variables. The other component,  $I$ , tracks what might have influenced the values of variables and is useful for dealing with integrity; we shall say that it tracks the *influencers* (or writers) of variables. It is natural to require that  $X \in R(x)$  and  $X \in I(x)$  whenever the variable  $x$  belongs to the process  $X$ , and an example security policy might be given by the following definition of  $R$  and  $I$ :

	a	b	c	d1	d2
$R$	A,D	B,D	C,D	D	D
$I$	A	B	C	A,B,D	B,C,D

**Typing the example.** To analyse the example we need to define  $R(x)$  and  $I(x)$  for all variables in such a way that they satisfy conditions imposed by a type system that are intended to ensure that the annotations are correct. In our extremely simple program there are two principles for ensuring this.

One concerns assignments of the form  $x:=y_1\#y_2$  where  $\#$  is one of the operators  $+$  or  $*$ . For confidentiality it is natural to impose that  $R(x) \subseteq R(y_1) \cap R(y_2)$ , or equivalently  $R(x) \subseteq R(y_1) \wedge R(x) \subseteq R(y_2)$ , because one should not allow any readers beyond those allowed by both  $y_1$  and  $y_2$ . For integrity it is natural to impose that  $I(x) \supseteq I(y_1) \cup I(y_2)$ , or equivalently  $I(x) \supseteq I(y_1) \wedge I(x) \supseteq I(y_2)$ , because one should not forget any of the influencers of  $y_1$  and  $y_2$ .

The other principle concerns assignments of the form  $x:=c$  where  $c$  is a constant. These are always acceptable. To fit the model of the previous case we might say that  $R(c) = \mathcal{U}$  and  $I(c) = \emptyset$  where  $\mathcal{U} = \{A, B, C, D\}$  is the universe of all processes and  $\emptyset$  is the empty set.

The definition of  $R$  and  $I$  expressed in the Table above satisfies the constraints imposed by our example program and is in line with the Decentralized Label Model [3].

**Non-Interference.** We would imagine that our parallel language is equipped with an operational semantics. Configurations might take the form  $\langle S, \sigma \rangle$  indicating that the system  $S$  of parallel processes is currently executing from the store  $\sigma$ . There would then be a transition relation  $\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$  indicating that one step of evaluation transforms  $\langle S, \sigma \rangle$  into  $\langle S', \sigma' \rangle$ . We do not have the space to present the details of this transition relation nor to discuss the possibility of labelling it (as is often done to deal with communication).

The non-interference approach to the semantic characterisation of a security policy then introduces the following notions to be defined below:  $S_1 \simeq S_2$  for when one system ( $S_1$ ) is similar to another ( $S_2$ ),  $\sigma_1 \cong \sigma_2$  for when one store ( $\sigma_1$ ) is close to another ( $\sigma_2$ ), and a notion of when a variable is low.

Unlike the case of bisimulations, the simulation relation  $\simeq$  is not necessarily reflexive, and the correctness of a type system amounts to ensuring that  $S \simeq S$  whenever the system  $S$  is admitted by the type system.

We now provide the definitions of the three notions introduced. The system  $S_1$  is similar to another  $S_2$ , written  $S_1 \simeq S_2$ , whenever  $\langle S_1, \sigma_1 \rangle \rightarrow \langle S'_1, \sigma'_1 \rangle$ ,  $\sigma_1 \cong \sigma_2$ , and  $\langle S_2, \sigma_2 \rangle \rightarrow \langle S'_2, \sigma'_2 \rangle$  ensure that  $S'_1 \simeq S'_2$  and  $\sigma'_1 \cong \sigma'_2$ , and vice versa. This definition is recursive and needs to be interpreted co-inductively in the usual manner of bisimulations.

The store  $\sigma_1$  is close to another  $\sigma_2$ , written  $\sigma_1 \cong \sigma_2$ , when they agree on all low variables:  $\sigma_1(x) = \sigma_2(x)$  whenever  $x$  is low. A variable  $x$  is said to be low if its security level  $L(x)$  (either  $R(x)$  or  $I(x)$  in our case) is dominated by some security value  $\ell$  (a subset of  $\mathcal{U}$  in our case) according to some partial order  $\sqsubseteq$  (being  $\supseteq$  for  $R$  and  $\subseteq$  for  $I$ ). The key property is that the set of low variables is closed under reducing the security classification under the partial order  $\sqsubseteq$ .

Most proofs of correctness [5] of a type system with respect to non-interference then rely on the type system ensuring that whenever  $y$  is somehow used in defining  $x$  (either explicitly or implicitly) then  $L(y) \sqsubseteq L(x)$ . This is fully in line with our explanation of typing the example above.

**Limitations of Non-Interference.** The above explanation uses lattice duality in sometimes choosing  $\sqsubseteq$  to be  $\subseteq$  and sometimes  $\supseteq$ . Let us rephrase the confidentiality component so that we can always use  $\subseteq$  for  $\sqsubseteq$ .

This amounts to replacing  $R(x)$  with its complement  $\bar{R}(x) = \mathcal{U} \setminus R(x)$ . The conditions imposed by typing are then changed to  $\bar{R}(c) = \emptyset$  whenever  $c$  is a constant and to demanding  $\bar{R}(x) \supseteq \bar{R}(y_1) \cup \bar{R}(y_2)$  for an assignment  $x := y_1 \# y_2$ . Intuitively,  $\bar{R}(x)$  lists those processes *not* allowed to read  $x$ . Our typing becomes:

	a	b	c	d	e
$\bar{R}$	B,C	A,C	A,B	A,B,C	A,B,C
$I$	A	B	C	A,B,D	B,C,D

In other words we have explicitly replaced the lattice for  $R$  with a dual lattice for  $\bar{R}$  which is unsurprising from a lattice theoretical point of view and quite in line with the usual statements of information flow that integrity is the dual of confidentiality (and used to motivate that technical developments often only focus on confidentiality).

*So what is the point?*

The point is that with this change the formal definition of non-interference is the same, symbol for symbol, for integrity as for confidentiality, and that there is not even the need to perform dual choices of the partial order.

*What does this mean?*

It means that while the non-interference result produces some validation of the type system against errors, it has no way of expressing whether or not  $I(x)$  denotes the set of influencers of  $x$ , or rather the set of processes not allowed to read  $x$ ; similarly for  $\bar{R}(x)$ . In other words:

Non-interference is unable to express the correctness of the intuitive explanations of what the security policies for influencers and readers are supposed to capture.

**Instrumented Semantics.** In our work with Airbus we are using ideas from program analysis in order to overcome the limitations of non-interference. In particular the use of an instrumented operational semantics where each transition is labelled with the *flow* taking place. In the case of an assignment statement performed by process  $Z$  we would have the following transition:

$$\langle x := y1 \# y2; S, \sigma \rangle \rightarrow_{(y1,x),(y2,x),(y1,Z),(y2,Z),(Z,x)} \langle S, \sigma[x \mapsto \sigma(y1) \# \sigma(y2)] \rangle$$

Here the subscript on the arrow indicates that both  $y1$  and  $y2$  are involved in producing  $x$ . Additionally we record that the process  $Z$  is reading  $y1$  and  $y2$  and is influencing (writing) the variable  $x$ . The full semantics would extend this to the other constructs in our parallel programming language and deal with both explicit (as illustrated) and implicit flow (not illustrated here).

In the full development we will allow policies to be influenced by the values of variables, so as to model content-dependent security policies. Much as in a Hoare logic [1] there would then be a policy  $(I, R)$  pertaining to the program point *before* the action and a possibly different policy  $(I', R')$  pertaining to the program point *after* the action.

The semantic correctness of the security policies with respect to a system  $S$  is then expressed by requiring that whenever

$$\langle S, \sigma \rangle \rightarrow^* \langle S', \sigma' \rangle \rightarrow_F \langle S'', \sigma'' \rangle$$

then we insist for the security policy  $(R, I)$  *before* the action and the security policy  $(R', I')$  *after* the action, that the following property

$$(R, I) \triangleright F \triangleleft (R', I')$$

holds. It is defined as follows:

- whenever  $(Z, x) \in F$  we have  $Z \in I'(x)$ ,
- whenever  $(y, Z) \in F$  we have  $Z \in R(y)$ , and
- whenever  $(y, x) \in F$  we have  $I(y) \subseteq I'(x)$  and  $R'(x) \subseteq R(y)$ .

This formulation makes it clear that constraints regarding influencers flow in the *forward* direction whereas constraints regarding readers flow in the *backward* direction. In this way we would be thinking of integrity as a forward analysis problem (like reaching definitions [4]) and confidentiality as a backward analysis problem (like live variables). This formulation clearly indicates the different directions of flow needed for formalizing integrity and confidentiality.

**Conclusion.** We have shown that non-interference falls short of providing convincing semantic explanations of the correctness of security policies for confidentiality and integrity as found in information flow frameworks like the Decentralized Label Model [3].

This contradicts conventional wisdom in the area of security policies for information flow. To quote an anonymous reviewer on a paper lacking a non-interference result: “My main complaint is the independence of the annotations from the actual semantics of the program and the non-interference properties it may have.”

This may be contrasted with the approach of static analysis where hardly any non-interference results are proved. To quote an international reviewer on a project attempting to establish such results: “Non-interference is a rather restrictive property so I am not totally convinced that one should start with it as a requirement.”

Our proposal therefore is to provide convincing semantic explanations of the correctness of security policies for confidentiality and integrity using suitably instrumented versions of an operational semantics.

**Acknowledgement.** We are supported by IDEA4CPS [2] and benefitted from discussions with Michael Paulitsch and Kevin Müller from Airbus.

## References

- [1] Krzysztof R. Apt. Ten years of Hoare’s logic: A survey - part 1. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.
- [2] IDEA4CPS: Foundations for Cyber-Physical Systems. Danish Research Foundations for Basic Research (Project DNRF86-10). Webpage: <http://idea4cps.dk>.
- [3] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *16th ACM Symposium on Operating Systems Principles*, pages 129–142, 1997.
- [4] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999. Second printing, 2005.
- [5] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

# Towards Component-based Reuse for Event-B

Andrew Edmunds<sup>1</sup>, Marina Walden<sup>1</sup>, and Colin Snook<sup>2</sup>

<sup>1</sup> Åbo Akademi University, Turku, Finland  
aedmunds@abo.fi, mwalden@abo.fi

<sup>2</sup> University of Southampton, UK cfs@ecs.soton.ac.uk

**Abstract:** An efficient re-use mechanism is a primary goal of many software development strategies; and is also important in the safety-critical domain, where formal development is required. Event-B can be used to develop safety-critical systems, but could be improved by development of a component-based re-use strategy. In this paper we outline a methodology, and the tool support required, for facilitating re-use of Event-B machines. As part of the ADVICeS project [10] we are seeking to improve re-use of Event-B artefacts. The creation of a library of components, and a way to assemble them, would facilitate this. We propose to extend iUML-B class diagrams [9], and extend the composition techniques introduced in [7], to allow specification of Event-B components, interfaces, and composite components. Initial investigation has been undertaken as part of the project ADVICeS, funded by Academy of Finland, grant No. 266373. The approach also addresses the need, in Event-B, for bottom-up scalability. We describe the process of creating library components, their composition, and specification of new properties (of the composed elements). We introduce the notion of Event-B components, component interfaces, and composite components. We describe the additional annotations, and discuss composition invariants.

## 1) Preliminaries

**Event-B** is a language and methodology [1, 2], with tool support provided by Rodin [3]. The system and its properties are specified using set-theory and predicate logic. It uses refinement [6] to show that the properties hold as the development proceeds. Refinement is used to add detail to the development. Event-B tools are designed to reduce the amount of interactive proof required during specification, and refinement steps [4]. Proof obligations (P.O.s) in the form of sequents, are automatically generated by the Rodin tool. The automatic prover can discharge many of the P.O.s. The remainder can be tackled with the interactive prover. Complex systems can be simplified using decomposition techniques [8].

The basic Event-B elements are *contexts*, *machines* and, *composed-machines*. Contexts define the static parts of the system using sets, constants and axioms. Machines describe the dynamic parts of a system using variables and events, and use invariant predicates to describe properties that should hold. We specify an event in the following way,

$$e \triangleq \text{ANY } p \text{ WHERE } G \text{ THEN } A \text{ END}$$

where  $e$  has parameters  $p$ ; a guarding predicate  $G$ ; and actions  $A$ . For the state updates (described in the action) to take place, the guard must be true.

**The Composition of Decomposed Machines:** Previous work [7] describes the composition that arises from the decomposition of a single machine. Multiple, decomposed sub-units, and the composed-machine construct, form a refinement of the abstract machine. We use the shared-event approach for decomposition, where the combined-events clause, of the composed-machine, refines an abstract event  $e$ . We write  $e_a \parallel e_b$  to combine events  $e_a$  and  $e_b$ , where subscripts  $a$  and  $b$  also identify the sub-units (machines). These events are said to *synchronize* (i.e., the events are enabled) when the conjunction of the guards are true. The combined actions are composed in parallel.



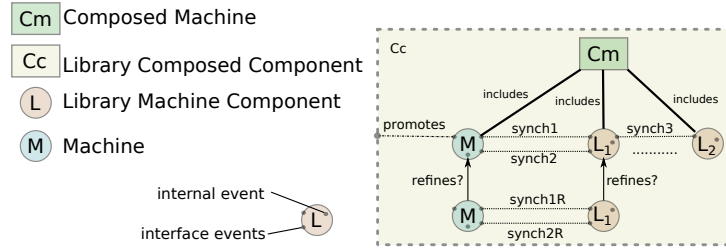


Figure 1: Machines “Included” in a Composition

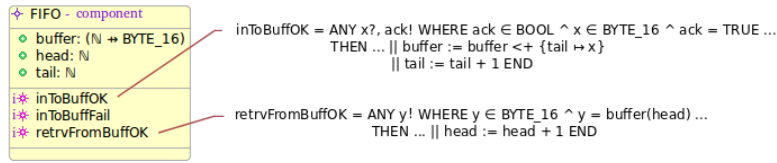


Figure 2: The FIFO Buffer Component

## 2) Composition with Components

By extending existing techniques, we aim to facilitate creation/use of a library of machines. Figure 1 shows a composed-machine  $Cm$  that includes library machine components  $L_1$  and  $L_2$ , and machines under construction  $M$ . Combined-events are shown using a dashed line between the machines. An interface reveals a set of events that may synchronize with some other machines, annotated with  $i$  against the event. See Fig. 2, an extended version of an existing iUML-B class diagram [9]. For parameter passing, the names and types of the *communicating* parameters are revealed, using  $?$  and  $!$  for input and output resp. A composed-machine may also be treated as a library component.

**Using Component Instances:** The component defined in Fig. 2 may be used to buffer data for producer and consumer models, see Fig. 3, a new diagrammatic representation in Event-B. Here, arrows represent associations, and dashed lines represent combined events.

**The Composition Invariant:** Each individual machine has its own set of invariants, and the composed-machine has *composition invariants* which specify properties about the composition. These properties cannot be specified in the individual machines. The composed machine needs visibility of *all* of the variables contained within the composition, and their included sets and constants. To ensure the composition invariant is satisfiable, we should add a guard  $G_{CI}$  to the composed event, but currently there is no feature in the tool to do this. The guards that preserve the new composition invariant can be added to the composed-machine’s combined event clause, subject to a tool enhancement. The guard will be added as follows,

$$e_a \parallel e_b \triangleq \text{ANY } p_a, p_b \text{ WHERE } G_{CI}(v) \wedge G_a \wedge G_b \text{ THEN } A_a \parallel A_b \text{ END}$$

In the example we may want the fifo buffer  $f1$  to hold odd numbers, and  $f2$  to hold even numbers. This is a property of the composition, and should be specified in the composition invariant clause. To specify this, we add an invariant, stating that all of the values in the producer’s  $f1$  buffers must have *mod* 2 of 1, and the values of the  $f2$  buffers must be *mod* 2 of 0; as in the following,

$$\forall p \cdot p \in \text{dom}(f1) \implies (\forall v \cdot v \in \text{ran}(\text{buffer}(f1(p))) \implies v \text{ mod } 2 = 1)$$

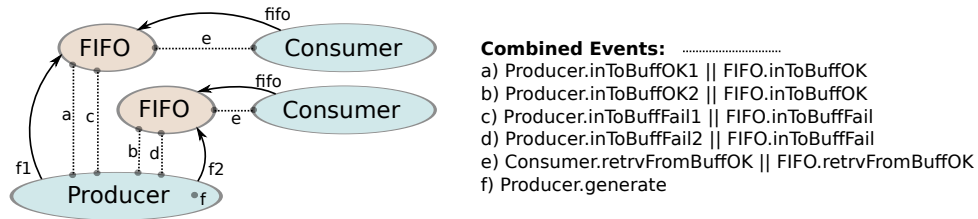


Figure 3: Component Instance Diagram

**Proof Obligations** We propose to take a *Design-By-Contract* (DBC) [5] view for input and output parameters. In our work the input and output parameters, and their type and direction information, form part of the interface specification. Using this, we can ensure that matching parameter’s output values fall within the range of the allowable inputs, by generating proof obligations.

### 3) Conclusions

In order to make the Event-B approach more flexible, we propose an extension to the existing composition approach, to introduce Event-B components. We add input, and output specifiers, “?” and “!” to event parameters, and extend iUML-B to describe components, and their interfaces. We add annotations, to identify externally visible events, while the remainder are hidden. We ensure communication is feasible, by generating additional precondition-style proof obligations; and provide a mechanism to add additional guards, to discharge the *composition invariant* proof obligations. We plan to investigate the use of components w.r.t. team-working. The parallel development of components, and artefacts within components, is key to making Event-B more agile.

## References

- [1] The Rodin User’s Handbook. Available at [http:// handbook.event-b.org/](http://handbook.event-b.org/).
- [2] J.R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
- [3] J.R. Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, November 2010.
- [4] S. Hallerstede. Justifications for the Event-B Modelling Notation. In J. Julliand and O. Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2007.
- [5] B. Meyer. Design by Contract: The Eiffel Method. In *TOOLS (26)*, page 446. IEEE Computer Society, 1998.
- [6] J. Wright R. Back. *Refinement Calculus: a systematic introduction*. Springer Science & Business Media, 2012.
- [7] R. Silva. *Supporting Development of Event-B Models*. PhD thesis, University of Southampton, May 2012.
- [8] R. Silva and M. Butler. Shared Event Composition/Decomposition in Event-B. In *FMCQ Formal Methods for Components and Objects*, November 2010. Event Dates: 29 November - 1 December 2010.
- [9] C. Snook and M. Butler. UML-B and Event-B: An Integration of Languages and Tools. In *The IASTED International Conference on Software Engineering - SE2008*, February 2008.
- [10] The ADVICeS Team. The ADVICeS Project. available at <https://research.it.abo.fi/ADVICeS/>.

# Improving Partial Order Reductions in Multithreaded Programs with Local First Search

Hernán Ponce de León<sup>1</sup>, Cristian Rosa<sup>2</sup>, and Keijo Heljanko<sup>1</sup>

<sup>1</sup> Helsinki Institute for Information Technology HIIT and Department of Computer Science, School of Science, Aalto University, Finland  
 {hernan.ponedeleon,keijo.heljanko}@aalto.fi

<sup>2</sup> CIFASIS, Rosario, Argentina  
 rosa@cifasis-conicet.gov.ar

Exploring the state space of a multithreaded program in an efficient manner is a fundamental problem in software verification. Concurrent transitions interleave in many ways quickly generating many equivalent but unequal states leading to the well known state space exposition problem. Two prominent approaches to deal with this problem are partial order reduction techniques (PORs) and unfoldings methods.

PORs techniques [1, 3, 4, 10] establish an equivalence relation between executions of the programs and explore a subset of all possible interleavings preserving at least one representative per equivalence class. At every state, they execute a subset of active or enabled transitions; those transitions can be computed statically [4] or dynamically [3]. Recently, an improvement to these methods have been proposed leading to an optimal PORs in the sense that exactly one execution is explored for each Mazurkiewicz trace [1]. All these approaches represent the possible executions of the program as a computation tree and prune some of its branching (once an equivalent branch has already been visited).

Unfoldings techniques [5, 6] models executions with partial orders together with a conflict relation to distinguish between different executions of the system. Both PORs and unfoldings techniques have shortcomings, but surprisingly, promising solutions for a given technique can be found in the opposite approach. For example PORs inexpensively add events to the current execution while computing possible extensions is the most demanding part of unfoldings techniques; on the other hand, explorations of repeated states and pruning of non-terminating executions is elegantly achieved in unfoldings with cut-off events.

The advantages of both approaches have been exploited together for the first time in [9] where they propose a technique that matches the test suite size of [1] but it explores an event structure rather than a computation tree. The former has a richer structure provided by a tree-like structure of partial orders. The use of partial order avoids the explicit enumeration of the order between concurrent or independent transitions of the program. The use of event structures suggests that improvements can be done to generate further reductions if we just intend to preserve local reachability [5].

Formally, an event structure is a tuple formed by a set of events  $E$ , a partially ordered relation  $\leq$  called causality (representing dependencies) and a symmetric and antireflective relation  $\#$  called conflict which is inherited w.r.t causality, i.e.  $e_1 \leq e_2$  and  $e_1 \# e_3$  implies  $e_2 \# e_3$ . Events not related by  $\leq$  or  $\#$  are called concurrent. The executions of an event structure are captured by its configurations, a causally-closed and conflict-free subset of events. Figure 1 shows the Hasse diagram of an event structure with nine events (transitive causalities or inherited conflict are removed for clarity); every event depends on  $\perp$ , e.g.  $\perp \leq 1$  and  $\perp \leq 4$  (since  $\leq$  is transitive); events 1 and 3 cannot belong to a same configuration since they are in conflict,

i.e.  $1 \# 3$ ; events 1 and 5 are concurrent. This event structure has four maximal configurations  $\{\perp, 1, 2, 5, 6\}$ ,  $\{\perp, 1, 2, 7, 8\}$ ,  $\{\perp, 3, 4, 5, 6\}$  and  $\{\perp, 3, 4, 7, 8\}$ .

The unfoldings semantics of a program can be expressed as an event structure [9]; while independent<sup>1</sup> transitions give rise to concurrent events, dependent ones generate causally dependent or conflicting events depending if they belong or not to the same execution. Figure 1 shows a program with 4 threads accessing two global variables  $x$  and  $y$ ; each pair of threads access a single variable by reading or writing it. Clearly the access to different variables is independent, thus  $x = 5 \diamond y = 1$ ,  $x = 5 \diamond c = y$ ,  $b = x \diamond y = 1$  and  $b = x \diamond c = y$ ; access to the same variable are dependent, i.e.  $x = 5 \diamond b = x$  and  $y = 1 \diamond c = y$ . The unfolding semantics of this program is given by the event structure of Figure 1. Each of the four maximal configuration corresponds to a deadlocking execution of the program. For example the configuration  $\{1, 2, 5, 6\}$  corresponds to the execution where variable  $x$  is written and then read followed by variable  $y$  being written and read.

Global variables:

```
int x, y = 0;
```

Thread 1:

```
local b = x;
```

Thread 2:

```
x = 5;
```

Thread 3:

```
local c = y;
```

Thread 4:

```
y = 1;
```

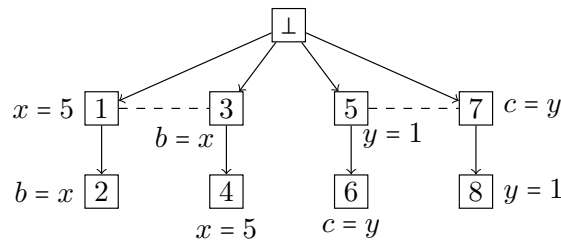


Figure 1: A multithreaded program and its unfolding semantics.

The optimality of [1, 9] states that no algorithm can explore less executions while preserving all Mazurkiewicz or deadlocking traces. If one instead targets at covering all local states of threads (which is sufficient to test for example local properties), it is not necessary any more to execute all Mazurkiewicz traces, but to cover every event of the event structure [5]. While computing the minimal set of executions to cover every local state is a very hard problem [8], we are interested at techniques that generate further reductions than those of [1, 9]. For the program generating the event structure of Figure 1, both optimal PORs explore four executions corresponding to the four maximal configurations. However if we are interested just in covering all the possible values local variables  $b$  and  $c$  might have, it is sufficient to explore only two executions, for example, all read transitions first in one execution and all write transitions first in another one. The first execution covers events 3,4,7,8 while the second one covers 1,2,5,6. To achieve this kind of reduction, we propose to use local first search [2, 7] on top of the unfolding-based PORs.

Local first search (LFS) was designed to optimize the search for local properties in transitions systems. The technique characterizes a restricted subset of traces that need to be explored to check local properties. For an event structure this means that only maximal events and their causal predecessors (those are called local or prime configurations) need to be explored. Since

<sup>1</sup>The independence relation arising from the program is denoted by  $\diamond$ ; while its complement (the dependence relation) is denoted by  $\diamonds$ .

the POR algorithms does not have complete information about the whole event structure (the event structure is constructed while the program is “being unfolded”), LFS performs an analysis to detect non prime configurations as soon as possible to avoid their exploration. This is based on a combinatorial aspect of the independence alphabet of the program. While the unfolding-based POR algorithm could explore the execution  $1 \cdot 5 \cdot 2 \cdot 6$ , we can detect (adding LFS) that the sub-configuration  $\{1, 5\}$  does not lead to a prime configuration and stop the exploration. Unfolding-based POR with LFS only explores the executions  $1 \cdot 2$ ,  $3 \cdot 4$ ,  $5 \cdot 6$  and  $7 \cdot 8$ . This approach explores shorter or smaller configuration, but still four executions are needed. However, it can be observed that those four configuration can be merged into, for example,  $1 \cdot 2 \cdot 5 \cdot 6$  and  $3 \cdot 4 \cdot 7 \cdot 8$ , but doing this during the exploration needs further algorithmics. This is similar to the problem of obtaining the minimal test suite to test a multithreaded program [8].

While [5] generates further reductions in the number of executions than the PORs techniques, it still relies on the construction of a Petri net unfolding and as such it suffers the computational cost of computing possible extensions. Since LFS can be used on top of the POR technique from [9], we believe this approach generates a good trade-off between the reduction in the size of the obtained test suite and the computational cost of the exploration.

## References

- [1] P. A. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal dynamic partial order reduction. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384, 2014.
- [2] S. Bornot, R. Morin, P. Niebert, and S. Zennou. Black box unfolding with local first search. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS'02, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2002.
- [3] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121, 2005.
- [4] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.
- [5] K. Kähkönen. *Automated Systematic Testing Methods for Multithreaded Programs*. Doctoral dissertation, School of Science, Aalto University, 2015.
- [6] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.
- [7] P. Niebert, M. Huhn, S. Zennou, and D. Lugiez. Local first search - A new paradigm for partial order reductions. In *12th International Conference on Concurrency Theory, CONCUR'01, Proceedings*, volume 2154 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2001.
- [8] H. Ponce de León, O. Saarikivi, K. Kähkönen, K. Heljanko, and J. Esparza. Unfolding based minimal test suites for testing multithreaded programs. In *15th International Conference on Application of Concurrency to System Design, ACSD 2015, Brussels, Belgium, June 21-26, 2015*, To appear.
- [9] C. Rodríguez, M. Sousa, S. Sharma, and D. Kroening. Unfolding-based partial order reduction. In *26th International Conference on Concurrency Theory, CONCUR'15, Proceedings*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. To appear.
- [10] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, pages 429–528, 1996.

# Modelling and analysis of normative contracts

John J. Camilleri and Gerardo Schneider

Chalmers University of Technology and University of Gothenburg, Sweden  
john.j.camilleri@cse.gu.se, gerardo@cse.gu.se

## 1 Introduction

Normative contracts are documents written in natural language, such as English or Swedish, which describe the permissions, obligations, and prohibitions of two or more parties over a set of actions, including descriptions of the penalties which must be payed when the main norms are violated. We encounter such texts frequently in the form of privacy policies, software licenses, and service agreements. These kinds of contracts are often long and difficult to follow for non-experts, and many people agree to such legally-binding documents without even reading them. Our goal is to provide front-end tools for analysing real-world contracts using formal methods. This involves a number of different components, ranging from entity extraction in natural language, the choice and design of a formalism for modelling contracts, textual and visual interfaces for working with contract models, query answering based on syntactic traversal and verification of temporal properties via conversion to timed automata.

## 2 Front-end

### 2.1 Extracting partial models

We have built a tool which takes a contract written in English and tries to extract various bits of information from it, in order to bootstrap the modelling process. It uses the Stanford parser [3] to produce dependency trees which we then analyse using some custom heuristics. This involves using the semantic relations between words to determine the subject, object, verb, modality and other elements from each sentence in the contract. After some manual post-editing, the tool's tabular output can be automatically converted to a model in our formalism. Our initial experiments show that the approach is already quite promising, both in terms of accuracy and in the reduction of effort involved for building a contract model.

### 2.2 Working with models

**Diagram editor** We visualise contract models as tree-like *C-O Diagrams* [4], an example of which can be seen in Figure 1. We have a web-based tool for working with these diagrams using a drag-and-drop interface along with real-time validation, in order to help the user build syntactically correct diagrams. The tool can import from and export to our XML-based interchange format COML.

**CNL** A controlled natural language (CNL) is a smaller, unambiguous and formally definable subset of a natural language. CNLs can be particularly useful for specific domains where the coverage of full language is not needed, or when it is possible to abstract away from some irrelevant aspects. We have defined a CNL for our contract models [2], implemented using the Grammatical Framework [6]. Figure 1b shows an example of what a contract clause looks like

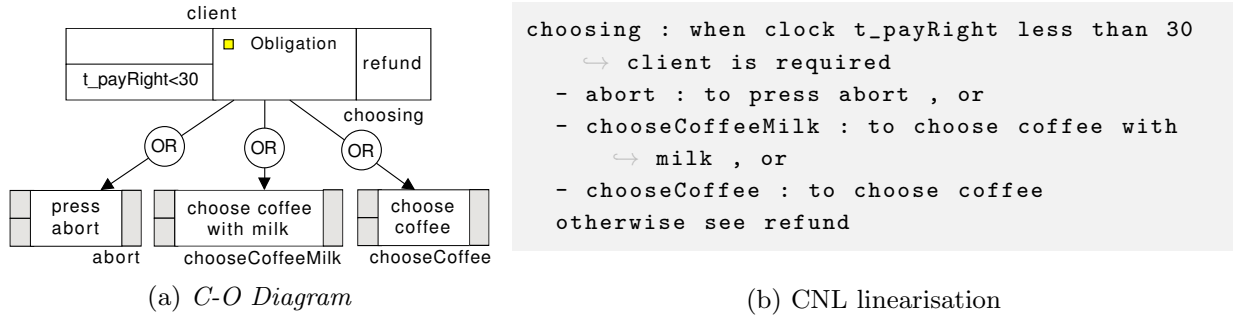


Figure 1: Example of visual and textual representations of an obligation clause.

in this language. As with the diagram editor, we also have a web-based CNL editing tool which comes with syntax checking, inline completion and basic highlighting to aid the user. It too can import and export to the COML format, enabling the user to switch back and forth between editing views.

All these tools are accessible at <http://remu.grammaticalframework.org/contracts/>.

### 3 Contract formalism

The formalism we use for modelling contracts is based on *C-O Diagrams* [4]. To this we have made a number of syntactic extensions, including a distinction between conditions for enactment and expiry, and the addition of generalised predicates as guards. Our largest contribution has been the definition of a completely novel trace semantics for *C-O Diagrams*, which formally defines what sequences of events can satisfy or violate a given model. We have also worked on a full back-end implementation in Haskell which, by parsing COML files into abstract contract models, can perform the kinds of analysis described in the following section.

### 4 Analysis

**Syntactic** Some queries can be checked at a syntactic level, such as identifying obligations without constraints or reparations. We introduce predicates over single clauses, which are the building blocks for defining syntactic properties. The predicate  $isObl(C)$  for example is true if the clause  $C$  is an obligation. Predicates may also take additional arguments, such as  $fromAgent(a, C)$ , which is true if agent  $a$  is responsible for clause  $C$ . Full queries can then be built out of these predicates, and a querying function returns the set of all clauses that satisfy the predicate. This function is defined inductively on the structure of contract models.

**Semantic** Syntactic analysis alone cannot be used to answer queries about the reachability of a given state. This requires taking into account the conditions applied to each clause, as well as a possible trace of previous events. These kinds of properties are computed using model checking. To do this, we convert contract models into networks of timed automata (NTA) [1] — finite state automata extended with guarded transitions, real-time clocks, and channel-based synchronisation between parallel automata (see example in Figure 2).

Using the UPPAAL tool [5], we can then test liveness and safety properties on our translated model using UPPAAL’s requirement specification language, which is a subset of TCTL including operators for *possibly* ( $E\Diamond$ ) *potentially always* ( $E\Box$ ) and *eventually* ( $A\Diamond$ ). This language allows

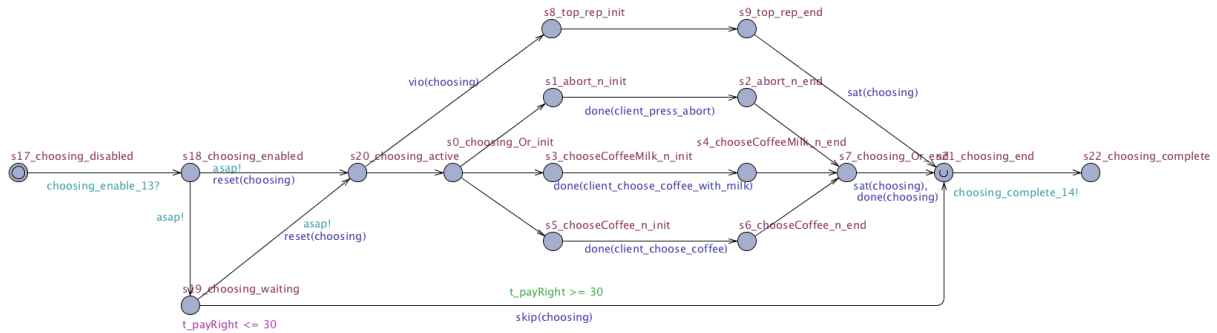


Figure 2: One of the automata produced from the translation of *C-O Diagram 1a*.

us to test for unexpected or undesirable situations which may potentially arise in the execution of a contract. These properties must be written and tested directly in UPPAAL by the user.

**Case studies** We have applied our methods to a few smaller case studies, including the terms of service for GitHub, Inc.<sup>1</sup> and a service-level agreement from hosting company LeaseWeb Inc.<sup>2</sup>

## 5 Future work

The largest piece missing from this work is the connection between high-level user questions in natural language and the low-level specification languages used for analysis. For this, we will define a query language (similar to the CNL described above) which can help users build properties for contract analysis by using a human-friendly interface. This will involve classifying the different queries we wish to allow, a method for converting these into logical properties using information from the translation into automata, and using the results of the analysis to produce properly formulated answers to the original query in natural language.

## References

- [1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [2] John J. Camilleri, Gabriele Paganelli, and Gerardo Schneider. A CNL for Contract-Oriented Diagrams. In *Controlled Natural Language*, volume 8625 of *LNCS*, pages 135–146. Springer, 2014.
- [3] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC 2006*, pages 449–454, 2006.
- [4] Gregorio Díaz, Maria Emilia Cambroner, Enrique Martínez, and Gerardo Schneider. Specification and Verification of Normative Texts using C-O Diagrams. *IEEE TSE*, 40(8):795–817, 2014.
- [5] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 2014.
- [6] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011. ISBN-10: 1-57586-626-9.

<sup>1</sup><https://github.com/site/terms>

<sup>2</sup><https://www.leaseweb.com/legal>



# From Explicit to Implicit Dynamic Frames in Concurrent Reasoning for Java (Extended Abstract)\*

Wojciech Mostowski

Center for Research on Embedded Systems, Halmstad University, Sweden  
wojciech.mostowski@hh.se

## 1 Introduction

In [9] we presented an approach to permission-based reasoning about concurrent Java programs in the context of the interactive program verifier KeY [1] which is based on Dynamic Logic and explicit dynamic frames [6, 13]. We argued for the *explicit* approach advocating the modular use (w.r.t. sequential vs. concurrent) and overall preciseness. It was noted, however, that changing our specification and verification approach to an established one of *implicit dynamic frames* (IDF) [11] should be also possible. In consequence, this would allow us to translate Separation Logic (SL) specifications [12, 2] into our framework to provide a powerful interactive theorem prover support for SL-like formalisms. In this context, we present some of the challenges associated with transition to implicit frames in KeY and possible solutions.

## 2 Permission-based Reasoning with Explicit Frames

As originally proposed in [6], the essence of specifying and reasoning about programs using explicit dynamic frames is the introduction of locations sets into the specification language as first class citizens and allowing them to be embedded within abstract predicates. In the KeY verification system which uses specially crafted Dynamic Logic for Java, this gives rise to JML\* specification language that introduces locations sets to the classic JML syntax [7] and the Dynamic Logic is equipped with means to reason about them. A classical, albeit minimalistic example of a Java program annotated with JML\* specification is shown in Fig. 1. The essential parts of this specification are frames, here expressed using abstraction through the JML model field `fp`. A read frame is specified with the **accessible** clause, and a write frame is specified with the **assignable** clause. As all other specifications, both clauses are effectively lemmas. Consequently, one is obliged to show that the lemma holds by showing that the corresponding method adheres to the limits of the frame, and then the lemma can be used to support proofs that involve the use of the method. For mutator methods the **assignable** clause is used to anonymise/havoc the corresponding locations of the program when the method is modularly applied in the proof to discharge a method call, while the **accessible** clause is used to establish the equivalence of two expressions under two different states when the expression is known,

```
public class ArrayList {
  Object[] cnt; int s;
  //@ model \locset fp = s, cnt, cnt[*];

  //@ ensures \result == s;
  //@ accessible fp;
  /*@ pure @*/ int size() { return s; }

  //@ ensures size() == \old(size()) + 1;
  //@ assignable fp;
  void add(Object o) { cnt[s++] = o; } }
```

Figure 1: A simple array list specified (incompletely) with explicit frames.

\*This work is supported by the Swedish Knowledge Foundation grant for the AUTO-CAAS project.

according to the read frame, not to depend on the locations changed between the two states. In our example, `size()` is guaranteed to always evaluate to the same value in all states in which `s` is not changed. In the Dynamic Logic for Java used by the KeY verifier suitable mechanisms are devised to both show the framing lemmas to hold and to use them in the proofs [13].

To verify concurrent behaviour, a convenient approach is to annotate programs with permission expressions, typically based on fractions [4], to guard every memory location access. A full permission grants a write access, while a partial permission grants only a read access. The construction of the verification method and the permission manipulation system guarantee that verified programs are data-race free. Typically, an SL-like framework is used for verification, among them IDF method of the Chalice verifier [8].

To enable permission-based reasoning in KeY without going too far away from its existing explicit frames framework, we add a second heap to track permissions and extend framing to that heap. In Fig. 2 a specification extended this way is shown. Most notably, heaps are now named explicitly in the specification (`heap` and `perms`) and are both given separate framing specifications. In most cases permission frames are actually empty, but not for methods and programming constructs that transfer permissions, e.g., mutual exclusion or sharing locks used to access another thread’s data. Additionally, in KeY we opted for developing a symbolic permission framework<sup>1</sup> [5] as an alternative to fractional permissions. The verification logic of KeY extends naturally to deal with the extra permission heap and methods are provided to enable fully modular and abstract specifications for the whole framework [9].

```
public class ArrayList {
  Object[] cnt; int s;
  //@ model \locset fp = s, cnt, cnt[*];

  //@ requires \readPerm(\perm(s));
  //@ ensures \result == s;
  //@ accessible<heap> fp;
  //@ accessible<perms> \nothing;
  /*@ pure @*/ int size() { return s; }

  //@ requires \readPerm(\perm(cnt));
  //@ requires \writePerm(\perm(s));
  //@ requires \writePerm(\perm(cnt[s]));
  //@ ensures size() == \old(size()) + 1;
  //@ assignable<heap> fp;
  //@ assignable<perms> \nothing;
  void add(Object o) { cnt[s++] = o; } }
```

Figure 2: Explicit frames specification with permissions.

### 3 Transformation to Implicit Dynamic Frames

In [11] it has been shown that IDF and SL are essentially equivalent w.r.t. expressiveness of specifications. Hence, here we concentrate on the task of treating IDF-style specifications in our framework.

The key observation in our explicit methodology is that because of the the use of permission annotations in specifications, particularly in preconditions, and verifying the code against these annotations, the **assignable** and **accessible** clauses are in essence obsolete. That is, the precondition provides the complete framing information for a given method – a read permission indicates that a method might be accessing a heap location (previously indicated in the **accessible** clause) and the write permission indicates that a method might be modifying a heap location (previously indicated in the **assignable** clause). More importantly, no heap location access (read or write) in the code would be allowed without a corresponding permission annotation, hence the permission annotations provide complete and sound framing specification. Dropping the frame specifications for the regular heap means two things. First, the frames do not have to be shown to hold in a separate proof obligation, the checking of the program w.r.t. the specified access

<sup>1</sup>Here the details of this permission system are not really relevant, the important part is that we can specify a permission to be a read or write permission, and that permissions can be transferred.

rights establishes the adherence to frame specification given by permission annotations. Second, it becomes a bit more difficult to apply modular method dispatch based on framing information, as the frames are not specified directly. The solution to this is to build the frame dynamically on demand using almost the same mechanism as we have presented in [9] to show self-framing of specifications w.r.t. permissions. For the assignable frame, a formula of the following shape is constructed:  $\text{pre} \wedge \forall_{o: \text{Object}, f: \text{Field}} (\text{writePerm}(o.f@perms) \rightarrow (o, f) \in \text{writeFrame})$ , where  $\text{writeFrame}$  is a fresh function symbol that collects all the heap locations for which we can show a write permission assuming that the method's precondition  $\text{pre}$  holds. The  $\text{writeFrame}$  can then be used in modular method dispatch.

However, the frames to the permission heap  $\text{perms}$  cannot be simply dropped in the same way without consequences for the specification method and patterns. The main reason is that the presence of a given permission in the specification does not, in general, imply that the permission heap  $\text{perms}$  is accessed or modified within the body of the method. In fact, the most common case is that a permission is present in the specification to allow a corresponding access on the regular heap, while the permission heap itself stays unchanged. Yet, assuming a frame for the  $\text{perms}$  heap as described above for the regular heap is the minimal sound approach if the frame is not to be stated explicitly. The resulting over-approximation of the permission frame can be mitigated on the specification level by specifying for each permission whether it is changed (and how) or not. In the latter case we propose to use a new keyword  $\backslash\text{samePerm}$ . Figure 3 shows the specification of the program in Fig. 2 modified to suit the implicit frame specification approach following the ideas just described. The need to specify all permissions in postconditions to enable precise reasoning is not surprising – all SL-like specifications are required to do so.

The implicit framing brings another small over-approximation issue. A write permission in the method's precondition implicates a corresponding location to be in the assignable clause of the method, while in reality the method might be only reading the location. Methods under-specified like this cannot be considered pure, despite being so. To check that this situation does not occur, an additional proof obligation in Java Dynamic Logic could be devised.

## 4 Conclusion

We presented the preliminary ideas for supporting IDF-style specifications in JML\* and the KeY program verifier for Java. As the explicit frames approach is deeply embedded in the KeY philosophy, the implementation considerations in KeY for IDF might bring further challenges. Moreover, we have not covered here interactions with JML\* *model methods* that we use for very flexible abstract and modular specifications in the context of inheritance [10]. Finally, despite the mentioned equivalence of IDF and SL, the ideas that we have discussed here are not sufficient for full and proper translation of SL specifications to JML\* and KeY logic. In particular, to fully support SL we also have to deal the separating conjunction operator  $*$  and the magic-wand operator  $-*$ , the latter being known for requiring non-trivial encodings [3].

```
public class ArrayList {
    Object[] cnt; int s;

    //@ requires \readPerm(\perm(s));
    //@ ensures \result == s;
    //@ ensures \samePerm(\perm(s));
    /*@ pure @*/ int size() { return s; }

    //@ requires \readPerm(\perm(cnt));
    //@ requires \writePerm(\perm(s));
    //@ requires \writePerm(\perm(cnt[s]));
    //@ ensures size() == \old(size()) + 1;
    //@ ensures \samePerm(\perm(s));
    //@ ensures \samePerm(\perm(cnt));
    //@ ensures \samePerm(\perm(cnt[s]));
    void add(Object o) { cnt[s++] = o; } }
```

Figure 3: IDF specification in JML\*.

## References

- [1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 8471 of *LNCS*, pages 1–17. Springer, 2014.
- [2] Afshin Amighi, Christian Haack, Marieke Huisman, and Clément Hurlin. Permission-based separation logic for multithreaded Java programs. *Logical Methods in Computer Science*, 11, 2015.
- [3] Stefan Blom and Marieke Huisman. Witnessing the elimination of magic wands. *International Journal on Software Tools for Technology Transfer*, pages 1–25, 2015.
- [4] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
- [5] Marieke Huisman and Wojciech Mostowski. A symbolic approach to permission accounting for concurrent reasoning. In *14th International Symposium on Parallel and Distributed Computing (ISPD 2015)*, pages 165–174. IEEE Computer Society, 2015.
- [6] Ioannis T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23:267–288, 2011.
- [7] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT*, 31(3):1–38, March 2006.
- [8] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, pages 195–222. Springer, 2009.
- [9] Wojciech Mostowski. Dynamic frames based verification method for concurrent Java programs. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, LNCS. Springer, 2015. To appear.
- [10] Wojciech Mostowski and Mattias Ulbrich. Dynamic dispatch for method contracts through abstract predicates. In *15th International Conference on MODULARITY*, pages 109–116. ACM, 2015.
- [11] Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. In Gilles Barthe, editor, *European Symposium on Programming*, volume 6602 of *LNCS*, pages 439–458. Springer, 2011.
- [12] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
- [13] Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in Java dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software Conference*, volume 6528 of *LNCS*, pages 138–152. Springer, 2011.

# A Formal Framework Supporting Unrestricted Software Changes in Object-Oriented Concurrent Systems\*

Olaf Owe, Jia-Chun Lin, and Ingrid Chieh Yu

Department of Informatics, University of Oslo  
E-mails: {olaf,kellylin,ingridcy}@ifi.uio.no

## Abstract

Program evolution may reveal bad design decisions, misunderstandings, or erroneous code and specifications. Problems made early may not be discovered until much later. Non-trivial changes of old code may be needed, and flexibility in making changes is essential. We propose a framework for reasoning about unrestricted program/specification changes, focusing on the challenges of concurrent and object-oriented programs.

## Motivation

Program development is in general a complicated process where many kinds of mistakes can be made over time. There can be bad design decisions, unclear specifications, misunderstandings, or erroneous code and specifications. Problems made early may not be discovered until much later. Redoing code made at an early stage in the software development may affect many parts of the overall system. Thus making changes in order to correct problematic decisions may create new problems that are hard to foresee. These kinds of problems are severe in the setting of concurrent programs when the interaction of the different concurrent units is complicated, and also in the setting of object-oriented programs, due to inheritance, dynamic binding and code reuse.

A systematic approach in which the consequences of a software change can be formalized, would be advantageous. Formal methods could be helpful in supporting specification and analysis of program properties. However, formal methods are mainly oriented towards developing correct specifications and programs rather than the process of redoing earlier decisions. It is therefore interesting to look at formal frameworks with support for unrestricted software changes, and such that the framework can detect possible consequences.

A trivial approach to reasoning about program changes is to re-verify and reprove all results whenever a change has been made. However this is time consuming and it is an expensive solution, especially for large software systems. Ideally we would like to reprove as little as possible, without losing soundness. And we would like to consider the setting of concurrent and object-oriented programs, which is both relevant in the software industry and challenging. The simplicity of a reasoning framework for software changes depends on the choice of specification and reasoning mechanisms as well as the language constructs and their semantics. For some programming paradigms, like shared variable concurrency, it is hard to analyze the effect of software changes, even with an advanced reasoning framework. We will therefore consider asynchronously communicating concurrent objects, since this setting offers compositional verification, and we consider program assertions over the communication history, since this captures all interactions and offers a measure of possible side-effects.

---

\*This work was done in the context of the EU projects H2020-644298 *HyVar: Scalable Hybrid Variability for Distributed Evolving Software Systems*, FP7-610582 *Envisage: Engineering Virtualized Services*, and FP7-ICT-2013-X *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations*.

## Related Work

In formal methods the notion of refinement is used to reflect software development. A refinement is in general leading from a design with certain properties to a design which preserves these properties, while adding more detail. In this way refinement is semantics-preserving. Certain refinement logics support the introduction of (additional) error values, thereby semantics is preserved as long as no errors appear. Banach et al have argued for the need of refinement-like steps that go beyond the limitation of semantics-preserving development [1]. However, their approach does not support analysis of program properties.

In the setting of object-oriented programs with inheritance, behavioral subtyping is the most common reasoning approach, restricting subclasses to obey the super-class specifications [7]. This means that subclasses must preserve behavior in some sense. Lazy behavioral subtyping [2] relaxes this condition; only behavior that is needed to verify local calls in a superclass must be respected by a subclass redefining the method. This gives added flexibility, allowing a larger class of changes without breaking the requirements. A number of works on asynchronously communicating concurrent objects, partly by the authors of this paper, consider certain forms of software and/or specification changes: The concept of dynamic software updates allows changes to superclasses [6]. Interface abstraction allows reasoning about remote calls to rely on the declared interface of the callee. This means that changes in a (super)class implementation may be done as long as the stated interface support is respected, and as long as subclass reasoning is not affected.

A calculus allowing changes to methods, (super)classes and interfaces is presented in [4]. The calculus can be seen as a generalization of lazy behavioral subtyping. Program properties, represented by Hoare triples, are classified in two categories for each class  $C$ , representing the verified ones and the unresolved (unverified) ones,  $\mathcal{U}(C)$ . The set of verified properties of a given class  $C$  and method  $m$  is denoted  $\mathcal{G}(C, m)$ . When the set of unverified program properties is verified (i.e.,  $\mathcal{U}(C)$  is empty) the class is found to be correct in the sense that all pre/post method specifications are satisfied by the corresponding implementation in a class as well as those in interfaces supported by the class. Changes in code or specifications may affect both categories. However, a program requirement added to  $\mathcal{U}(C)$  may be impossible to verify (in case the Hoare triple is not satisfied), and it will then remain in  $\mathcal{U}(C)$ , and there is no guarantee that this problem is detected.

The approach in [3] addresses transformation of classes and allow classes in the middle of a class hierarchy to be changed. Modifications are archived by means of update operations *modify* and *simplify*. The modify operations extend class definitions, allowing code such as new fields, method definitions, guarantees, and interfaces to be added to classes, and existing methods to be redefined. The simplify operations allow redundant methods to be removed from class definitions. The approach does not classify classes using  $\mathcal{G}$  and  $\mathcal{U}$  such as in [4], rather, for each update applied to a class, all verification work is done to methods affected by the update. However, the approach is limited to interface additions, i.e., implemented interfaces can not be removed or modified and methods can only be changed if the guarantees are strong enough to satisfy the constraints of the implemented interfaces. And superclass requirements needed to handle local calls are imposed on subclasses, as in [4].

Another line of works consider proof reuse, including partial reuse of proofs of earlier verified properties. This may require some storage of proof outlines or non-trivial verification steps. This means that when a (part of a) program is corrected, one may try to rerun previous proofs to alleviate the verification burden [9]. The notion of abstract method calls allows reuse of abstract proof outlines, for a fixed method body, while their instances may need further work when other methods or requirements are changed [5] These approaches simplify the verification

task of evolving programs. The amount of proof reuse can be balanced against the amount of automation. Efficiently automated proof need not be reused while interactive proofs could benefit from reuse, if possible. In our approach we will be oriented towards a language with a high degree of automation of verification conditions and proof reuse is therefore not in our focus. For simplicity we will therefore not consider techniques for proof reuse.

## Approach

In this paper we try to find a good framework for reasoning about unrestricted program changes, focusing on the challenge of concurrent and object-oriented programs. We use interface abstraction since this limits the visible effect of low-level changes. Furthermore we allow multiple invariants for each class/interface since this makes it meaningful to add new invariants for old classes upon need. In program evolution, specifications should be allowed to evolve. Finally we consider primitives for changing old or new interfaces or classes, including methods, invariants, inheritance, and interface support. In general a software modification will consist of a sequence of several primitive changes constituting a meaningful modification. We assume type correctness, and therefore fields may only be removed when no longer in use. Thus one must modify all methods and invariants using a field before removing it.

Our primitives allow unrestricted changes of code (assuming type correctness). This means that one may write combinations of code and invariants that are inconsistent, for instance when a class does not satisfy the requirements of its interface(s). The framework will help in detecting such inconsistencies so that they may be resolved. In order to determine the consequences of changes in a (super)class the framework needs to keep track of dependencies of local calls. In particular for reasoning about inheritance, we build on the approach of *behavioral interface subtyping* [8] where each class is only required to satisfy its invariant(s) and interface specifications and any other local specifications given in the class. This means that a method redefined in a subclass may break the requirements of the superclass, even the minimal requirements imposed in the case of lazy behavioral subtyping. This opens up for more liberal modifications than earlier work based on lazy behavioral subtyping as no superclass requirements are imposed on a subclass. The consistency of a class is determined by looking at the class itself, its interface(s), and any reused code from superclasses. For a software modification one must first determine the affected code, and for each class containing such code one must re-verify the affected parts.

## References

- [1] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Engineering and theoretical underpinnings of retrenchment. *Science of Computer Programming*, 67(2&A53):301 – 329, 2007.
- [2] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.
- [3] J. Dovland, E. B. Johnsen, O. Owe, and I. C. Yu. A proof system for adaptable class hierarchies. *Journal of Logical and Algebraic Methods in Programming*, 84(1):37 – 53, 2015.
- [4] J. Dovland, E. B. Johnsen, and I. C. Yu. Tracking behavioral constraints during object-oriented software evolution. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 253–268. Springer, 2012.
- [5] R. Hähnle, I. Schaefer, and R. Bubel. Reuse in software verification by abstract method calls. In M. P. Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2013.

- [6] E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In M. Steffen and G. Zavattaro, editors, *Proc. 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of Lecture Notes in Computer Science, pages 15–30. Springer, June 2005.
- [7] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
- [8] O. Owe. Verifiable programming of object-oriented and distributed systems. In L. Petre and E. Sekerinski, editors, *From Action System to Distributed Systems: The Refinement Approach*. CRC Press, 2015. To appear.
- [9] W. Reif and K. Stenzel. Reuse of proofs in software verification. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 284–293. Springer-Verlag, 1993.



# An implementation in Agda of Sutner's decision algorithms for injectivity and surjectivity of one-dimensional cellular automata

Silvio Capobianco<sup>1,2</sup> and Niccolò Veltri<sup>1,3</sup> \*

<sup>1</sup> Institute of Cybernetics at TUT

<sup>2</sup> Email: `silvio@cs.ioc.ee`

<sup>3</sup> Email: `niccolo@cs.ioc.ee`

We discuss an Agda formalization of the algorithms, due to Klaus Sutner [3], that decide injectivity and surjectivity of one-dimensional *cellular automata* by equating these properties with features of cycles in specific finite labeled graphs.

A cellular automaton (briefly, CA) is a quadruple  $\mathcal{A} = \langle d, Q, \mathcal{N}, f \rangle$  where the *dimension*  $d \geq 1$  is an integer, the *alphabet*  $Q$  is finite and contains at least two elements, the *neighborhood*  $\mathcal{N} \subseteq \mathbb{Z}^d$  has  $N \geq 1$  elements  $\nu_1, \dots, \nu_N$ , and the *local update rule*  $f$  is a function from  $Q^s$  to  $Q$ . The *global transition function* of the CA  $\mathcal{A}$  on the set  $\mathcal{C} = Q^{\mathbb{Z}^d}$  of  $d$ -dimensional *configurations* is defined as the synchronous application of the local update rule to each point of  $\mathbb{Z}^d$ , according to the formula

$$F_{\mathcal{A}}(c) = \lambda x : \mathbb{Z}^d. f(c(x + \nu_1), \dots, c(x + \nu_N)) \quad \forall c \in \mathcal{C}. \quad (1)$$

For  $d = 1$  one can always take  $\mathcal{N} = \{m, \dots, m + N - 1\}$  for suitable  $m \in \mathbb{Z}$ , and see  $f$  as a function of the word  $q_1 \cdots q_N$  corresponding to the concatenation of its arguments  $q_1, \dots, q_N$ .

Deriving properties of a CA's global transition function exclusively from its finitary formulation is, in general, undecidable. There are, however, important exceptions: in dimension 1, both injectivity and surjectivity of (1) are decidable. This is due to the *Garden of Eden theorem* ruling that a CA is surjective if and only if distinct configurations only differing in finitely many points have distinct images, and the characterization of injective 1D CA as those that are injective on the set of *periodic* configurations. (See [1] for an introduction to CA theory.)

Recall that the product of two labeled graphs  $G_1 = (V_1, E_1, \mathcal{L})$  and  $G_2 = (V_2, E_2, \mathcal{L})$  with the same set  $\mathcal{L}$  of labels is the graph  $G = (V, E, \mathcal{L})$  where  $V = V_1 \times V_2$  and  $((x_1, x_2), (y_1, y_2)) \in E$  with label  $\ell \in \mathcal{L}$  if and only if  $(x_1, y_1) \in E_1$  and  $(x_2, y_2) \in E_2$  both with label  $\ell$ .

For  $N \geq 2$ , the *de Bruijn graph* of order  $N$  on the alphabet  $Q$  is the graph  $G = (V, E)$  where  $V = Q^{N-1}$  and  $(u, v) \in E$  if and only if  $u = xw$  and  $v = wy$  for suitable  $x, y \in Q$  and  $w \in Q^{N-2}$ . If  $\mathcal{A} = \langle 1, Q, \mathcal{N}, f \rangle$  is a 1D CA with alphabet  $Q$  and neighborhood  $\mathcal{N} = \{m, \dots, m + N - 1\}$ , we can label  $xwy \in Q^N$  with  $f(xwy) \in Q$ : we call the *Sutner graph* of the CA the product of the labeled graph so obtained with itself. By our discussion above, calling *diagonal* the subgraph generated by the pairs  $(w, w)$  with  $w \in Q^{N-1}$ , the following hold [3]:

1. A one-dimensional cellular automaton is injective if and only if no cycle in its Sutner graph touches a node outside the diagonal.
2. A one-dimensional cellular automaton is surjective if and only if no cycle in its Sutner graph joins the diagonal with the outside.

---

\*This research was supported by the ERDF funded projects EXCS and Coinduction, the Estonian Ministry of Education and Research institutional research grant IUT33-13, and the Estonian Science Foundation grant no. 9398.

For the present work, which is still in progress, we use version 2.4.2.3 of the Agda programming language [5] with version 0.9 stable of the standard library. Agda is a dependently-typed functional programming language based on intuitionistic type theory. The standard library is rich in algebra and category theory, but only deals with acyclic graphs. As a side project, we start the development of a small Agda library for graphs, following what is done by [2] in Haskell: such library would, in our aims, include an efficient implementation of the depth-first search algorithm. The choice of Agda is motivated by the greater expressiveness of its type system, which allows not only to implement algorithms, but also to prove their correctness.

Our module `1DCA` is parameterized by three values  $\mathbf{q} \ \mathbf{m} \ \mathbf{s} : \mathbb{N}$ . We implement the alphabet as the initial interval  $\mathbf{Q}$  of the natural numbers with  $\mathbf{succ} \ \mathbf{q}$  elements, the neighborhood as an interval of starting point  $-\mathbf{m}$  and size  $\mathbf{N} = \mathbf{succ} (\mathbf{succ} \ \mathbf{s})$ , and the local update rule  $\mathbf{f}$  as a function of type  $\mathbf{Vec} \ \mathbf{Q} \ \mathbf{N} \rightarrow \mathbf{Q}$ . We define patterns as vectors on  $\mathbf{Q}$ .

Configurations are implemented as pairs of *streams* (defined coinductively) over a given alphabet, with the convention that the configuration  $(\dots, q_{-2}, q_{-1}, q_0, q_1, \dots)$  is represented by the ordered pair of streams  $((\mathbf{q}_{-1}, \mathbf{q}_{-2}, \dots) \rightarrow | (\mathbf{q}_0, \mathbf{q}_1, \dots))$ : the constructor of the type `Conf` of configurations is  $\rightarrow |$ , rather than the standard comma, to emphasize the role of the point  $0 \in \mathbb{Z}$ . Equality is defined through bisimilarity in the standard way. Such method allows to define *translations* (i.e., the functions  $\sigma_t = \lambda c : \mathcal{C}. (\lambda x : \mathbb{Z}^d. c(x + t))$ ) straightforwardly. A pattern  $\mathbf{p}$  is turned into a periodic configuration by a function `periodic =  $\lambda \mathbf{p} . \mathbf{behind} \ \mathbf{p} \rightarrow | \mathbf{ahead} \ \mathbf{p}$` , where `ahead  $\mathbf{p}$`  is the periodic stream obtained by concatenating  $\omega$  copies of  $\mathbf{p}$ , and `behind  $\mathbf{p}$`  the one similarly obtained from the reverse of  $\mathbf{p}$ .

In order to update entire configurations by the global transition function, we must be able to update single points by the local update rule: having done this, we can corecursively apply the procedure to both the `ahead` and `behind` component of the configuration. The global transition function is defined *up to translations*: as the latter are bijections, this does not alter injectivity and surjectivity of the CA—which is our present focus.

The de Bruijn graph is implemented via its edge relation. From this, we construct the Sutner graph: two pairs of vectors  $(\mathbf{x} \ \mathbf{s}, \mathbf{x}' \ \mathbf{s}')$ ,  $(\mathbf{y} \ \mathbf{s}, \mathbf{y}' \ \mathbf{s}')$  of length  $\mathbf{succ} \ \mathbf{s}$  are related if and only if  $(\mathbf{x} \ \mathbf{s}, \mathbf{y} \ \mathbf{s})$  and  $(\mathbf{x}' \ \mathbf{s}', \mathbf{y}' \ \mathbf{s}')$  are both in the de Bruijn relation, and in addition  $\mathbf{f}$  takes the same value on the words of length  $\mathbf{N}$  corresponding to the two pairs. A path in the Sutner graph will then be the reflexive and transitive closure of the Sutner relation; a cycle, a *nonempty* path with the same endpoints; a loop, a cycle of length 1.

As a cycle of length 2 or more always belongs to a strongly connected component, and every strongly connected component with two or more nodes always contains a cycle (possibly a loop), the Sutner conditions for injectivity and surjectivity can be tested by Tarjan's strongly connected components algorithm [4] as follows:

1. The CA is injective if and only if the Sutner graph has no loops outside the diagonal, and no strongly connected components of size 2 or more that contain a point outside the diagonal.
2. The CA is surjective if and only if the Sutner graph has no strongly connected components that contain nodes both inside and outside the diagonal.

For condition 1 we can exploit that every loop in the Sutner graph is on a node of the form  $(q^{N-1}, p^{N-1})$  with  $q, p \in \mathbf{Q}$ .

We can prove in Agda that, if the global transition function is injective, then all cycles in the Sutner graph are contained in the diagonal. The proofs of the converse of the above, and of the corresponding statements for surjectivity, are currently being implemented.

Future work will deal with formalizations of cellular automata theory. In particular, we conjecture that the aforementioned Garden of Eden theorem can be formalized in Agda: which would allow to prove the other direction with regard to surjectivity.

## References

- [1] Kari, J. (2005) Theory of cellular automata: A survey. *Theor. Comput. Sci.* **334**, 3–33.
- [2] King, D. and Launchbury, J. (1994) Lazy depth-first search and linear graph algorithms in Haskell. Glasgow Workshop on Functional Programming 1994. doi:10.1.1.45.3876
- [3] Sutner, K. (1991) De Bruijn graphs and linear cellular automata. *Complex Systems* **5**, 19–31.
- [4] Tarjan, R.E. (1972) Depth-first search and linear graph algorithms. *SIAM J. Comput.* **1(2)**, 146–160.
- [5] The Agda Wiki. [wiki.portal.chalmers.se/agda/](http://wiki.portal.chalmers.se/agda/)

# Rule formats for bounded nondeterminism in nominal structural operational semantics<sup>\*†</sup>

Álvaro García-Pérez

ICE-TCS, School of Computer Science, Reykjavík University, Iceland  
alvarog@ru.is

## 1 Introduction

Structural operational semantics (SOS) [14,15] is a widely used formalism for defining the formal semantics of computer programs and for proving properties of the corresponding programming languages. In the SOS formalism a transition system specification (TSS) [9], which consists of a signature together with a set of inference rules, specifies a labelled transition system (LTS) [11] whose states (i.e., processes) are closed terms over the signature and whose transitions are those that can be proved using the inference rules.

Rule formats [3,13] are syntactically checkable restrictions on the inference rules of a TSS that guarantee some useful property of the associated LTS. We focus on the finiteness of the number of outgoing transition from a given state, which is referred to as bounded nondeterminism in [18]. Broadly, bounded nondeterminism is taken as a synonym of finite branching [7]. Finite branching breaks down into the more elementary properties of initials finiteness and image finiteness [1].

Nominal structural operational semantics (NoSOS) [5] enriches the SOS formalism by adopting the nominal techniques from [8,16] to deal with names and variable-binding operations within the SOS framework. The nominal techniques allow one to extend pleasantly structural induction and recursion to languages with variable-binding operations, without the need to redo on a case-by-case basis a large number of routine constructions that deal with renaming of bound variables [8]. The NoSOS framework develops the nominal techniques in the general setting of meta-theory of SOS [3,13] and makes them applicable to a wide variety of specific languages.

The investigations on rule formats for bounded nondeterminism are far from being new. Vaandrager [17] introduced a rule format for SOS based on the de Simone format [6] that guarantees that the associated LTS is finite branching. Following Vaandrager, Bloom [4] introduced a rule format for his GSOS formalism that also guarantees a finite-branching LTS. Finally, Fokkink and Vu [7] introduced yet another less restrictive rule format for SOS which adapts the notion of strict stratification from [10], and showed that a TSS in this format induces an LTS that is finite branching.

Our work takes this programme further by contributing on three fronts:

- (i) We provide syntactic conditions that use *global* information to filter more junk rules (i.e., rules that are never involved in a proof tree) than the conditions of the rule format in [7], which uses *local* information.

---

<sup>\*</sup>Joint work with Luca Aceto, Ignacio Fábregas and Anna Ingólfssdóttir.

<sup>†</sup>This research has been supported by the project ‘Nominal Structural Operational Semantics’ (nr. 141558-051) of the Icelandic Research Fund.

- (ii) We extend the applicability of the rule formats to the nominal setting by tackling one of the most prominent challenges there, namely that of allowing variables to occur in the actions that label a transition [2].
- (iii) We consider a family of bounded-nondeterminism properties that are more elementary than finite branching, and which include image finiteness and initials finiteness [1].

The examples that follow are representative of each of these three contributions. Recall from [7] that an LTS is finite branching iff for every process  $p$ , the set  $\{(l, p') \mid p \xrightarrow{l} p'\}$  is finite.

**Example 1.1.** *Let the signature  $\Sigma$  consist of unary function symbols  $f$  and  $g$  and constant symbol  $c$ . Let the set  $L$  consist of label  $l$ . Consider the TSS*

$$\frac{}{g(x) \xrightarrow{l} x} \qquad \frac{g^i(x) \xrightarrow{l} x}{f(x) \xrightarrow{l} x}, \quad i \in \mathbb{N}$$

where  $g^i$  stands for applying  $i$  times the function symbol  $g$  to its argument. The TSS induces an LTS that is finite branching. Notice that for  $g(p)$  and  $f(p)$ , with  $p$  any process, the only provable transitions are respectively  $g(p) \xrightarrow{l} p$  and  $f(p) \xrightarrow{l} p$  since the axiom on the left allows one to instantiate the rule template on the right only for  $i = 1$ .

Previous work on rule formats for finite branching [7] introduces the  $\eta$ -types, which determine an over-approximation of the set of rules that give rise to transitions. One of the conditions of the rule format in [7] is to require the  $\eta$ -types to be finitely inhabited. The  $\eta$ -typing discipline is *local* and thus it is not strong enough to discern whether the instances of the rule template of Example 1.1 would take part in a proof tree or not. For all  $i \in \mathbb{N}$ , the instances of the rule template above have one and the same  $\eta$ -type, which is infinitely inhabited. This renders the TSS of Example 1.1 out of the conditions of the rule format.

**Example 1.2.** *Consider the nominal TSS (NTSS for short) for term-for-atom substitution ( $a \mapsto x$ ) on page 6 of [5], which includes the rule ( $\text{abs1}_{\text{TS}}$ ) that we reproduce next:*

$$\dots \quad \frac{x \xrightarrow{y \mapsto z} x' \quad a \# z \quad a \# y}{[a]x \xrightarrow{y \mapsto x} [a]x'} \quad (\text{abs1}_{\text{TS}}) \quad \dots$$

Recall from [5] that the nominal term  $[a]x$  is an abstraction where the atom  $a$  is abstracted in the nominal term  $x$ , and that  $a \# z$  is a freshness assertion that is provable iff the atom  $a$  does not appear free in the nominal term  $z$ . The substitution of term  $x$  for atom  $a$  in term  $t$  is modelled as an LTS with transitions  $t \xrightarrow{a \mapsto x} t\{a \mapsto x\}$ . Notice that  $\cdot \mapsto \cdot$  is a binary function symbol and that the labels include arbitrary nominal terms, i.e., the  $y \mapsto z$  that labels the first premiss of rule ( $\text{abs1}_{\text{TS}}$ ) contains variables  $y$  and  $z$ .

The rule format in [7] requires the labels of transitions to be ground. The occurrence of variables is important in order to determine whether a proof tree can introduce spurious variables, which could be unified to any term and could possibly break bounded nondeterminism. Besides, the proof of correctness of the rule format in [7] relies on the TSS having a *strict stratification* (see Definition 4 of [7]) that entails an order relation among processes and enforces

the sources of the premisses in a rule to be less than the source of the rule. This prevents the associated LTS to implement unguarded recursion. The occurrence of variables in the labels opens for the possibility of the LTS to break bounded nondeterminism. In order to regain boundedness, a suitable notion of strict stratification may have to take the labels into account, which poses a non-trivial challenge.

Recall from [1] that an LTS is initials finite iff for every process  $p$  the set  $\{l \mid \exists p' \text{ s.t. } p \xrightarrow{l} p'\}$  is finite, and it is image finite iff for every process  $p$  and every label  $l$ , the set  $\{p' \mid p \xrightarrow{l} p'\}$  is finite. An LTS is finite branching iff it is both initials finite and image finite, and thus the latter properties are more elementary than the property of finite branching.

**Example 1.3** (Example 5.3 of [7]). *Let  $r \in \mathbb{R}_{>0}$ . Consider the operator for deadlock in real-time Basic Process Algebra [12], which can be expressed by the rule*

$$\frac{}{\delta[r] \xrightarrow{\delta[s]} \checkmark} \quad 0 < s < r.$$

*Process  $\delta[r]$  is infinitely branching and has an uncountable set of initials. However,  $\delta[r]$  has a finite set of images since for a given time  $s$  the only possible transition labelled by  $\delta[s]$  is  $\delta[r] \xrightarrow{\delta[s]} \checkmark$ . The associated LTS is image finite, but it is not finite branching nor initials finite.*

The  $\eta$ -types of [7] constrain the cardinality of the actions that label the premisses in a rule. In conjunction with the conditions that ensure that a proof tree cannot introduce spurious variables in the targets of transitions, the  $\eta$ -types being finitely inhabited and the strict stratification are enough to guarantee finite branching. However, finite branching is only one among many bounded-nondeterminism properties, and it is certainly not the most elementary. For the property of initials finiteness, the process  $p$  is fixed and the rule format constrains the cardinality of the labels  $l$  in transitions  $p \xrightarrow{l} p'$  while allowing the targets  $p'$  to be unbounded. For the property of image finiteness, the process  $p$  and the label  $l$  are fixed and the rule format constrains the cardinality of the targets  $p'$  in transitions  $p \xrightarrow{l} p'$ . In order to guarantee these elementary properties, more refined syntactic conditions than the ones in [7] are needed.

## 2 Contributions

**Filtering junk rules.** We introduce the  $S$ -types that, differently from the  $\eta$ -types in [7], rely on the *global* information provided by the order relation between processes that is entailed by the strict stratification. The  $S$ -types filter out those rules for which the sources of the premisses and the sources of the rule are not in the order relation. For instance, the TSS of Example 1.1 has a strict stratification given by

$$\begin{aligned} S(f(p)) &= 1 \\ S(g(p)) &= 0. \end{aligned}$$

An instantiation of the rule template on the right of Example 1.1 has source  $f(x)$  and premiss with source  $g^i(x)$  for some  $i \in \mathbb{N}$ , which unify respectively with processes  $f(p)$  and  $g^i(p)$  (with  $p$  any process). The strict stratification  $S$  is undefined for  $g^i(p)$  with  $i \neq 1$ , and thus the source of the rule and the source of the premiss are in the order relation only when  $i = 1$ , i.e.,  $S(g(p)) < S(f(p))$ . The other instantiations of the rule template do not have a valid  $S$ -type and can be disregarded because they will never take part in a proof tree. The  $S$ -type of the instantiation where  $i = 1$  is finitely inhabited.

**Variables in labels and elementary properties.** We introduce a transformation for transitions that turns the *triadic* representation  $p \xrightarrow{l} p'$  into a *dyadic* representation  $o \rightarrow d$  ( $o$  for *origin* and  $d$  for *destination*) that places the label  $l$  either in the origin, i.e.,  $(p, l) \rightarrow p'$ , or in the destination, i.e.,  $p \rightarrow (l, p')$ . (As in [2], we assume that nominal terms are closed under Cartesian product.) The rule format is applied verbatim to the dyadic representation of an NTSS. The transformation serves two purposes:

- (i) The 0-fold Cartesian product (unit) is the new *administrative* label in all transitions, while the *real* labels are arbitrary terms either in the origin or in the destination. This circumvents duly the concerns about the proof trees introducing spurious variables and about the strict stratification.
- (ii) Different dyadic transformations enforce different bounded-nondeterminism properties, i.e.,  $p \rightarrow (l, p')$  for finite branching and  $(p, l) \rightarrow p'$  for image finiteness. (More on initials finiteness below.) For example, the dyadic transformation for image finiteness of rule  $(\text{abs1}_{\text{T}_S})$  of Example 1.2 reads:

$$\frac{(x, y \xrightarrow{T} z) \rightarrow x' \quad a \# z \quad a \# y}{([a]x, y \xrightarrow{T} x) \rightarrow [a]x'}$$

The LTS for term-for-atom substitution is initials finite because for a given nominal term  $t$  and label  $a \xrightarrow{T} x$  ( $a$  is the atom to be substituted for and  $x$  is the subject of the substitution) there is only one result of the substitution, i.e.,  $t \xrightarrow{a \xrightarrow{T} x} t\{a \mapsto x\}$ .

In order to ensure initials finiteness, we relax certain syntactic conditions of the rule format for finite branching as to unconstrain the cardinality of targets  $p'$  in the destinations  $(l, p')$ . We refer to this relaxation of the rule format as *extrusion*.

**Other bounded-nondeterminism properties.** The dyadic transformation for finite branching applies to yet another elementary bounded-nondeterminism property. By extruding the labels instead of the targets the following property is ensured: for every process  $p$ , the set  $\{p' \mid \exists l \text{ s.t. } p \xrightarrow{l} p'\}$  is finite.

Other dyadic transformations in which the label  $l$  is itself the origin or the destination are possible, which we have dubbed  $l \uparrow (p, p')$  and  $(p, p') \downarrow l$ . Together with extrusion, our rule format affords for a family of up to eight bounded-nondeterminism properties based on the dyadic transformations, which include finite branching, initials finiteness and image finiteness. We are not aware whether any of the five other properties has received any particular name in the literature. These properties may have relevance in the nominal setting where labels of transitions have a prominent role.

### 3 Future work

We have considered NTSSs *after stripping away freshness assertions*. We conjecture that, for such NTSSs, the freshness assertions can only restrict the cardinality of provable transitions from *infinite* (all) to *cofinite* (all but finitely many), and hence freshness assertions do not have an impact in the bounded-nondeterminism properties. Proving this result is still work in progress.

## References

- [1] Samson Abramsky. *Domain theory and the logic of observable properties*. PhD thesis, Department of Computer Science, Queen Mary College, University of London, 1987.
- [2] Luca Aceto, Matteo Cimini, Mohammad Reza Mousavi, Michel A. Reniers, and Murdoch James Gabbay. Nominal SOS. To be submitted.
- [3] Luca Aceto, Wan Fokkink, and Chris Verhoef. Structural operational semantics. In A. Ponse J.A. Bergstra and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 3, pages 197–292. Elsevier, 2001.
- [4] Bard Bloom. CHOCOLATE: Calculi of Higher Order COmmunication and LAMBda TERms (preliminary report). In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of the 21st ACM Symposium on Principles of Programming Languages, Portland, Oregon*, pages 339–347. ACM Press, 1994.
- [5] Matteo Cimini, Mohammad Reza Mousavi, Michel A. Reniers, and Murdoch James Gabbay. Nominal SOS. *Electronic Notes in Theoretical Computer Science*, 286:103–116, 2012.
- [6] R. de Simone. Higher-level synchronising devices in MELJE–SCCS. *Theoretical Computer Science*, 37(3):245–267, 1985.
- [7] Wan Fokkink and Thuy Duong Vu. Structural operational semantics and bounded nondeterminism. *Acta Informatica*, 39(6-7):501–516, 2003.
- [8] M. J. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Symposium on Logic in Computer Science, Trento, Italy*, pages 214–224. IEEE Computer Society Press, 1999.
- [9] J. F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, 1992.
- [10] Jan Friso Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118(2):263–299, 1993.
- [11] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.
- [12] A. S. Klusener. *Models and axioms for a fragment of real time process algebra*. PhD thesis, Department of Mathematics and Computing Science, Technical University of Eindhoven, 1993.
- [13] Mohammad Reza Mousavi, Michel A. Reniers, and Jan Friso Groote. SOS formats and meta-theory: 20 years after. *Theoretical Computer Science*, 373(3):238–272, 2007.
- [14] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981.
- [15] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
- [16] Christian Urban, Andrew Pitts, and Murdoch J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.
- [17] F. Vaandrager. Expressiveness results for process algebras. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop on Semantics: Foundations and Applications, Beekbergen, The Netherlands*, volume 666 of *Lecture Notes in Computer Science*, pages 609–638. Springer, 1993.
- [18] Rob J. van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In F. J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany*, volume 247 of *Lecture Notes in Computer Science*, pages 336–347. Springer, 1987.



# Winning Cores in Parity Games

Steen Vester

Technical University of Denmark, Kgs. Lyngby, Denmark  
stve@dtu.dk

## Abstract

Whether parity games can be solved by a polynomial-time algorithm is a well-studied problem which has not yet been resolved. In this talk we propose a new direction for approaching this problem based on the novel notion of a winning core.

We give two different, but equivalent, definitions of a winning core and show a number of interesting properties about them. This includes showing that winning cores can be computed in polynomial time if and only if parity games can be solved in polynomial time and that computation of winning cores is in the intersection of NP and co-NP.

We also present a deterministic polynomial-time approximation algorithm for solving parity games based on computing winning cores. It runs in time  $O(d \cdot n^2 \cdot m)$  where  $d$  is the number of colors,  $n$  is the number of states and  $m$  is the number of transitions. The algorithm returns under-approximations of the winning regions in parity games. It works remarkably well in practice as it solves all benchmark games from the PGSOLVER framework in our experiments completely and outperforms existing algorithms in most cases. Correctness of the output of the algorithm can be checked efficiently.

## 1 Introduction

Solving parity games [1] is an important problem of both theoretical and practical interest. This is known to be in  $\text{NP} \cap \text{co-NP}$  [2] and  $\text{UP} \cap \text{co-UP}$  [7] but in spite of the development of many different algorithms (see e.g. [13, 18, 8, 17, 9, 15]), frameworks for benchmarking such algorithms [6, 10] and families of parity games designed to expose the worst-case behaviour of existing algorithms [8, 4, 5] it has remained an open problem whether a polynomial-time algorithm exists.

Various problems for which polynomial-time algorithms are not known can be reduced in polynomial time to the problem of solving parity games. Among these are model-checking of the propositional  $\mu$ -calculus [11, 3, 16], the emptiness problem for parity automata on infinite binary trees [14, 2] and solving boolean equation systems [12].

Some of the most notable algorithms from the literature of solving parity games include the recursive algorithms from [13, 18] using  $O(n^d)$  time, the small progress measures algorithm [8] using  $O(d \cdot m \cdot (n/d)^{d/2})$  time, the strategy improvement algorithm [17] using  $O(n \cdot m \cdot 2^m)$  time, the big step algorithm [15] using  $O(m \cdot n^{d/3})$  time and the dominion decomposition algorithm [9] using  $O(n^{\sqrt{n}})$  time. Here,  $n$  is the number of states in the game,  $m$  is the number of transitions and  $d$  is the maximal color occurring in the game.

## 2 Contributions

First, we introduce some notation. In the following we fix a finite parity game  $\mathcal{G}$  (for a definition, see e.g. [18]) with colors in  $\{1, \dots, d\}$ . The set of winning states for player  $j$  in  $\mathcal{G}$  is denoted  $W_j(\mathcal{G})$ . We say that a (finite or infinite) sequence  $\rho = s_0 s_1 \dots$  of states with at least one transition is *0-dominating* if the greatest color  $e = \max\{c(s_i) \mid i > 0\}$  occurring in a non-initial state of  $\rho$  is even and *1-dominating* if it is odd. Examples are shown in Figure 1.

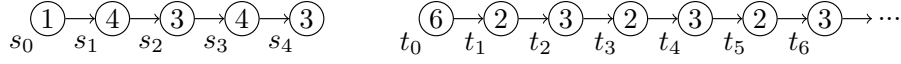


Figure 1: The sequence to the left is 0-dominating and the sequence to the right is 1-dominating.

We say that a play  $\rho$  begins with  $k$  consecutive  $j$ -dominating sequences if there exists indices  $i_0 < i_1 < \dots < i_k$  with  $i_0 = 0$  such that  $\rho_{i_\ell} \rho_{i_\ell+1} \dots \rho_{i_{\ell+1}}$  is  $j$ -dominating for all  $0 \leq \ell < k$ . This definition is straightforwardly extended to an infinite number of consecutive  $j$ -dominating sequences. As examples, the sequence on the left in Figure 1 begins with two consecutive 0-dominating sequences  $s_0 s_1$  and  $s_1 s_2 s_3$  whereas the sequence to the right begins with only one 0-dominating sequence  $t_0 t_1$ , but not two consecutive 0-dominating sequences.

For a player  $j$  and a parity game  $\mathcal{G}$  the *winning core*  $A_j(\mathcal{G})$  is defined as the set of states in  $\mathcal{G}$  from which player  $j$  can force that the play begins with an infinite number of consecutive  $j$ -dominating sequences. Our main results on winning cores are the following.

**Proposition 1.** *Let  $\rho$  be a play. Then  $\rho$  begins with an infinite number of consecutive  $j$ -dominating sequences if and only if  $\rho$  is  $j$ -dominating and winning for player  $j$ .*

**Theorem 1.** *Let  $\mathcal{G}$  be a parity game and  $j$  be a player. Then*

1.  $A_j(\mathcal{G}) \subseteq W_j(\mathcal{G})$
2.  $A_j(\mathcal{G}) = \emptyset$  if and only if  $W_j(\mathcal{G}) = \emptyset$

**Proposition 2.** *There exists a parity game  $\mathcal{G}$  where  $A_j(\mathcal{G})$  is not a  $j$ -dominion [9].*

**Theorem 2.** *Computing winning cores is in  $\text{NP} \cap \text{co-NP}$ .*

**Theorem 3.** *Computing winning cores can be done in polynomial time if and only if parity games can be solved in polynomial time.*

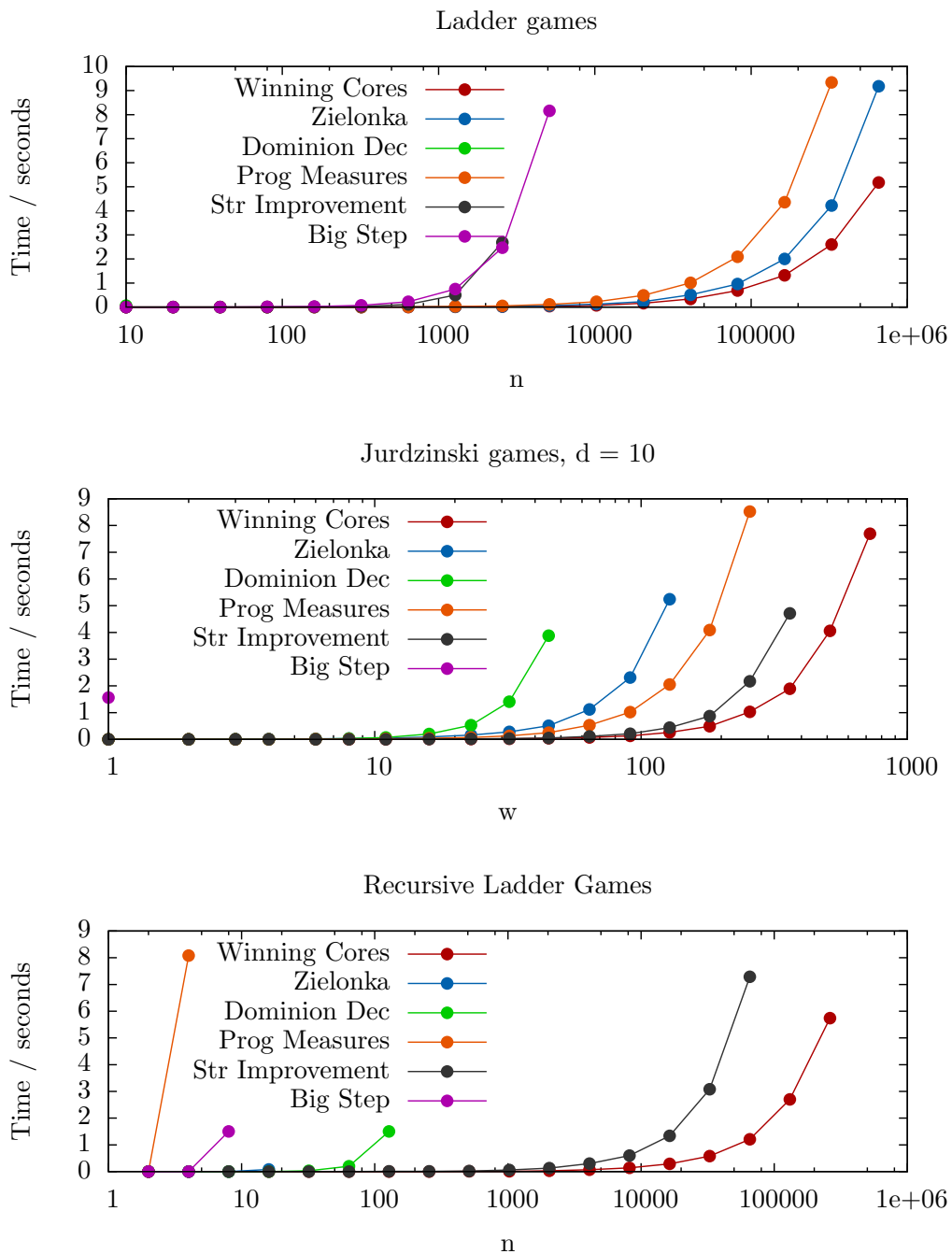
Proposition 1 gives us two equivalent definitions of the same concept which is not immediately obvious. Theorem 1 provides us with valuable information about the winning cores and, further, it is used to design an algorithm for solving parity games based on computing winning cores. Proposition 2 is interesting as many algorithms for solving parity games focus on finding  $j$ -dominions whereas the winning core is a subset of winning states that is not necessarily a  $j$ -dominion. Finally, Theorem 2 and 3 make the search for a polynomial-time algorithm for computing winning cores a viable direction in the search for a polynomial-time algorithm for solving parity games.

The results are used to develop a new fixpoint algorithm that calculates under-approximations of the winning regions in parity games. This algorithm runs in time  $O(d \cdot n^2 \cdot m)$  and is very fast in practice. Further, it can be efficiently checked whether the output of the algorithm is correct. This means that it can be applied with confidence when it outputs the correct result.

In our experiments the algorithm performs remarkably well returning the complete winning region in most cases. In Figure 2 are experimental results on correctness in randomly generated games. Further, the algorithm returns the correct results on all other benchmark games from the PGSolver framework on which it has been tested. A comparison of running times for some of the benchmarks can be seen below. These benchmarks are games designed to be difficult for some of the existing algorithms to solve. Our algorithm has been implemented in OCaml as a part of the PGSolver framework using the same basic data structures as the other algorithms.

$n \setminus b$	$d = 4$			$d = \lceil \sqrt{n} \rceil$			$d = n$		
	2	5	10	2	5	10	2	5	10
100	0.002%	0.000%	0.000%	0.114%	0.001%	0.000%	0.559%	0.011%	0.000%
1000	0.000%	0.000%	0.000%	0.571%	0.000%	0.000%	2.113%	0.000%	0.000%

Figure 2: Ratio of games where the algorithm did not return the entire winning region.  $n$  is the number of states,  $d$  is the number of colors and  $b$  is the out-degree. For every fixed  $n$ ,  $d$  and  $b$  the experiments were done on 100.000 games generated randomly by PGSolver [6].



## References

- [1] E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 368–377, 1991.
- [2] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. On model checking for the  $\mu$ -calculus and its fragments. *Theor. Comput. Sci.*, 258(1-2):491–522, 2001.
- [3] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In Albert Meyer, editor, *Proceedings of the First Annual IEEE Symp. on Logic in Computer Science, LICS 1986*, pages 267–278. IEEE Computer Society Press, June 1986.
- [4] Oliver Friedmann. An exponential lower bound for the parity game strategy improvement algorithm as we know it. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 145–156, 2009.
- [5] Oliver Friedmann. Recursive algorithm for parity games requires exponential time. *RAIRO - Theor. Inf. and Applic.*, 45(4):449–457, 2011.
- [6] Oliver Friedmann and Martin Lange. Solving parity games in practice. In *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*, pages 182–196, 2009.
- [7] Marcin Jurdzinski. Deciding the winner in parity games is in  $UP \cap co-UP$ . *Inf. Process. Lett.*, 68(3):119–124, 1998.
- [8] Marcin Jurdzinski. Small progress measures for solving parity games. In *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Lille, France, February 2000, Proceedings*, pages 290–301, 2000.
- [9] Marcin Jurdzinski, Mike Paterson, and Uri Zwick. A deterministic subexponential algorithm for solving parity games. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 117–123, 2006.
- [10] Jeroen J. A. Keiren. Benchmarks for parity games. *CoRR*, abs/1407.3121, 2014.
- [11] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.
- [12] Angelika Mader. *Verification of Modal Properties Using Boolean Equation Systems*. Edition versal 8. Bertz Verlag, Berlin, Germany, 1997. Imported from DIES.
- [13] Robert McNaughton. Infinite games played on finite graphs. *Ann. Pure Appl. Logic*, 65(2):149–184, 1993.
- [14] Andrzej Włodzimierz Mostowski. Regular expressions for infinite trees and a standard form of automata. In *Computation Theory - Fifth Symposium, Zaborów, Poland, December 3-8, 1984, Proceedings*, pages 157–168, 1984.
- [15] Sven Schewe. Solving parity games in big steps. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference, New Delhi, India, December 12-14, 2007, Proceedings*, pages 449–460, 2007.
- [16] Colin Stirling. Local model checking games. In *CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21-24, 1995, Proceedings*, pages 1–11, 1995.
- [17] Jens Vöge and Marcin Jurdzinski. A discrete strategy improvement algorithm for solving parity games. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 202–215, 2000.
- [18] Wiesław Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.

# Privacy in Evolving Social Networks

Raúl Pardo<sup>1</sup> and Gerardo Schneider<sup>2</sup>

<sup>1</sup> Dept. of Computer Science and Engineering, Chalmers, Sweden.

<sup>2</sup> Dept. of Computer Science and Engineering, University of Gothenburg, Sweden  
 {pardo@chalmers.se, gerardo@cse.gu.se}

## 1 Motivation

Over the past decade, the use of the social networks like Facebook and Twitter, just to mention two of the most popular ones, has increased at the point of becoming ubiquitous. Many people access *Social Networks Services* (SNSs) on a daily basis; e.g. to read the news, share pictures with their friends or check upcoming events. Nearly 70% of the Internet users are active on SNSs, as shown by a recent survey [4]. Empirical studies have also shown that the current privacy protections offered by SNSs are very far from the users' expectations [6, 5]. One of their weaknesses is the inability for users to express desirable privacy policies. This is because, the privacy settings offered by SNSs are too coarse-grained. Furthermore, many users are not fully aware of the result of activating a privacy policy or if the policy protects their personal data as they expect. We believe citizens should be in power to control and decide on how much information to make public. One way to do so is by providing users with means to define their own privacy policies and give guarantees that they will be respected. Privacy in SNSs may be compromised in different ways: from direct observation of what is posted (seen by non-allowed agents), by inferring properties of data (*metadata privacy leakages*), indirectly from the topology of the SN (e.g., knowing who our friends are), to more elaborate intentional attackers such as *sniffers* or *harvesters* [3]. Among others, one of the origins of these attacks comes from their privacy enforcement mechanism, the so called Relationship-Based Access Control (ReBAC) [2].

We aim at developing a privacy enforcement mechanism which offers social network users the possibility of expressing finer-grained privacy policies, enabling them to deal with (certain kinds of) implicit disclosure of sensitive information. Moreover this mechanism should take into account that the social network *evolves* and enforce the privacy policies as events occur. We have developed the privacy policy framework  $\mathcal{FPPF}$  for social networks [8, 7], which is briefly described in next section.

## 2 The First-Order Privacy Policy Framework $\mathcal{FPPF}$

$\mathcal{FPPF}$  is composed by a static part which describes the state of the social network at a given point in time, and the dynamic part, which models how the social network evolves as events occur [7]. The components are:

**A social network model.** SNM is a *social graph*, a graph whose nodes represent users, and edges represent different kind of relationships between users. The graph is enriched with information on the knowledge the users of the social network have, and what they are permitted to do.

**A knowledge-based logic.**  $\mathcal{KBL}_{\mathcal{SN}}$  is an epistemic logic including a permission operator, which provides the possibility to reason about what the agents know and what they are allowed to do. The logic allows us not only to access and reason about the explicit knowledge of an agent, but also about implicit knowledge (through inferences).

**A formal privacy policy language.**  $\mathcal{PPL}_{SN}$  is a language for writing privacy policies for each individual user.

**A labelled transition system for social networks.**  $\mathcal{LTS}_{SN}$  contains a set of SNMs, which result from the execution of events in the SNSs. These events as operational semantics rules, which are divided in the following categories: i) *Epistemic*. These rules describe how the knowledge and the permission change in the SNM. ii) *Social topology*. These rules modify the social topology of the SNM, i.e. the users and their relationships. For instance, adding new users, relationships between them, etc. iii) *Policy*. These rules allow for the modification of the set of privacy policies of the agents. iv) *Hybrid*. These are rules which combine changes of any of the categories above.

Besides, the framework also comes with a *satisfaction relation* defined for the logic  $\mathcal{KBL}_{SN}$ , and a *conformance relation* defined for the policy language  $\mathcal{PPL}_{SN}$ . The framework may be tailored by providing suitable instantiations of the different relationships, the events, the atomic predicates representing what is to be known, and the additional facts or rules a particular social network should satisfy.

In order to show how  $\mathcal{FPPF}$  can be used, we have instantiated the privacy policies of Facebook and Twitter [8, 7], which are two of the most used social networks nowadays. For instance, one of Facebook's privacy policies is responsible for setting the audience of a post, where the user can choose among 'Friends', 'Only me' and 'Custom'. In  $\mathcal{FPPF}_{\text{Facebook}}$  it would be split in 3 policies. In the mentioned instantiation if  $u$  wants the audience of her posts to be her Friends, it would be written as follows:  $\llbracket \neg S_{Ag \setminus \text{friends}(u) \setminus \{u\}} \text{post}(u, j, n) \rrbracket_u$  where  $S_G \varphi$  is a formula stating "somebody in the group  $G$  knows  $\varphi$ ",  $Ag$  is the set of all the agents in the SNM,  $u, j \in Ag$ ,  $n \in \mathbb{N}$ ,  $\text{post}(u, j, n)$  represents post  $n$ , written by  $u$  and posted in  $j$ 's timeline and  $\text{friends}(u)$  is an function which returns all the friends of  $u$ .

As we mentioned before  $\mathcal{FPPF}$  is an generic framework, therefore we could combine instantiations of two (or more) different social networks in one. This is a very useful and innovative feature, since currently it is becoming more common to connect several accounts from different social networks and share information between them. As a final example of the use of our framework, we present below an example of a privacy policy concerning the combination of  $\mathcal{FPPF}_{\text{Twitter}}$  and  $\mathcal{FPPF}_{\text{Facebook}}$  [8]. The following privacy policy: *Only my friends in Facebook who are following me in Twitter can know my location* will be written in our formalism as  $\llbracket \neg S_{Ag \setminus (\text{friends}(u) \cap \text{Followers}(u)) \setminus \{u\}} u.\text{location} \rrbracket_u$ . The rules defining how the SNM evolves are given using small step operational semantics. For example, in Twitter the most basic event is called *tweet*. It is used to share a 140 characters long message with a set of users. Let  $\text{tweet}(tu, \text{TweetInfo}, \text{Audience})$  be the event representing that user  $tu$  shares  $\text{TweetInfo}$  to the set of users  $\text{Audience}$ . The following rule models the behaviour of the event,

$$\frac{\forall \varphi \in \text{TweetInfo}, \forall i \in \text{Audience } KB'_i = KB_i \cup \{\varphi\}}{SN \xrightarrow{\text{tweet}(tu, \text{TweetInfo})} SN'}$$

We use  $SN \xrightarrow{\text{tweet}(tu, \text{TweetInfo})} SN'$  to denote that an SNM  $SN$  evolves to a new SNM  $SN'$ .  $\text{TweetInfo}$  is a set of formulae in  $\mathcal{KBL}_{SN}$ , which represents the content of the message, e.g. if the predicate  $\text{age}(u) \in \text{TweetInfo}$ , it means that  $u$ 's age is part of the message.  $KB_i$  represents the *knowledge base* of a user  $i$ . In SNMs knowledge bases are used to store all the information that the users know. Let  $KB'_i$  be the knowledge base of  $i$  in  $SN'$  and analogously for  $KB$ , then  $\forall \varphi \in \text{TweetInfo}, \forall i \in \text{Audience } KB'_i = KB_i \cup \{\varphi\}$  means that after the execution of the event *tweet*, all users part of the audience will *know* all the information shared in the tweet.

In [7] we define what means for a SNS to be privacy-preserving. Specifically, we say if all privacy policies are executed before and after the execution of any event in the SNS, then the

SNS is privacy-preserving. We also proved that Twitter is privacy-preserving. Additionally, we proved that adding new desirable policies to Twitter and Facebook make the SNSs not privacy-preserving.

### 3 Current and Future Work

Traditionally the semantics of “the logic of knowledge” or epistemic logic are given by means of Kripke structures, where the *uncertainty* of the agents is modelled [1]. However, in  $\mathcal{FPPF}$  we explicitly model the concrete knowledge of the agents. While these approaches to model knowledge seem to be complementary, we recently found out that it is possible to encode SNM to the equivalent Kripke structure. Nevertheless, this encoding entails some issues that we are currently investigating. For instance, the properties of the binary relations in the Kripke structure affect the implicit knowledge that agents can infer. It might be possible that some properties of knowledge that hold in Kripke structures do not hold in SNMs. Besides, we plan to investigate whether the models are equivalent, i.e. we claimed that it is possible to encode SNMs in Kripke structures, but we do not know if the opposite is possible.

Besides, we are currently implementing a prototype of our framework in the open source SNS *Diaspora*\*<sup>12</sup>. We have extended *Diaspora*\* with several privacy settings, which are not currently offered in other major SNS (including Facebook or Twitter). We aim at implementing the full power of  $\mathcal{FPPF}$ . A centralised implementation of the enforcement mechanism would create a huge bottleneck, since SNSs are massively distributed and millions of events could be triggered at the same time. Therefore, we are currently looking into distributed architectures for monitoring algorithms, which can help us to monitor the privacy policies of all users in the SNS efficiently.

As future work we plan to extend  $\mathcal{FPPF}$  to support real time policies. For example, a user could write a policy saying “My boss cannot know my location between 20:00-23:00” or “The audience of the post on my timeline during my birthday is only my friends”.

## References

- [1] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about knowledge*, volume 4. MIT press Cambridge, 2003.
- [2] P. W. Fong. Relationship-based access control: Protection model and policy language. In *CO-DASPY'11*, pages 191–202. ACM, 2011.
- [3] B. Greschbach, G. Kreitz, and S. Buchegger. The devil is in the metadata - new privacy challenges in decentralised online social networks. In *PerCom Workshops*, pages 333–339. IEEE, 2012.
- [4] A. Lenhart, K. Purcell, A. Smith, and K. Zickuhr. Social media & mobile internet use among teens and young adults. *Pew Internet & American Life Project*, 2010.
- [5] Y. Liu, K. P. Gummadi, B. Krishnamurthy, and A. Mislove. Analyzing facebook privacy settings: User expectations vs. reality. *IMC '11*, pages 61–70. ACM, 2011.
- [6] M. Madejski, M. Johnson, and S. Bellovin. A study of privacy settings errors in an online social network. (*PERCOM Workshops'12*), pages 340–345.
- [7] R. Pardo, M. Balliu, and G. Schneider. A formal approach to preserving privacy in social networks (extended version). Technical report, Chalmers University of Technology.
- [8] R. Pardo and G. Schneider. A formal privacy policy framework for social networks. In *SEFM'14*, volume 8702 of *LNCS*, pages 378–392. Springer, 2014.

<sup>1</sup><https://diasporafoundation.org/>

<sup>2</sup><https://github.com/raulpardo/ppf-diaspora>

# Formally Verifying Exceptions for Low-level code with Separation Logic

Marco Paviotti and Jesper Bengtson

IT University of Copenhagen  
Rued Langaards Vej 7, 2300, Denmark  
{mpav, jebe}@itu.dk

## Abstract

Exceptions in low-level architectures are implemented as synchronous interrupts: upon the execution of a faulty instruction the processor jumps to a piece of code that handles the event. Previous work has shown that assembly programs can be written, verified and run using higher-order separation logic [2]. However, execution of faulty instructions is then under specified by either being undefined or terminating with an error. In this work, we initiate the study of synchronous interrupts and prove an example of memory allocator, thus showing that it is possible to write positive specifications of programs that fault. All of our results are mechanised in the interactive proof assistant Coq.

## 1 Introduction

Assembly code is difficult to prove correct. Standard Hoare-logics make implicit assumptions about the control flow of programs and assume that the code  $c$  in a triple  $\{P\}c\{Q\}$  has one entry point and one exit point, even though it may internally contain loops and method calls. In assembly programs we cannot make this assumption as the control flows of these languages are inherently unstructured. Control flow is altered primarily by two mechanisms – jump commands and interrupts. Jump commands allow developers to execute code stored nearly anywhere in memory; their use is an active choice, much like writing a loop or calling a method. Interrupts, on the other hand, occur either when something has gone catastrophically wrong (such as dividing by zero or reading from un-mapped memory) or when an action from the environment requires processing (such as the user pressing a key, a change to the file system is made, or the processor clock ticks). While some of the aspects of interrupts might resemble that of function calls, there are substantial differences: synchronous interrupts are not called explicitly but are dependent on a certain events that can occur at run-time, secondly, there cannot be infinitely many calls as after three interruptions the machine reboots. These interrupts are typically referred to as synchronous. Another denotation for synchronous interrupts is exceptions, due to their similarity with the exceptions encountered in languages like Java or ML, and we will use the terms interchangeably.

In this paper we extend Kennedy et al.’s semantics for the x86 machine [3] and Jensen et al.’s [2] program logic by adding support for synchronous interrupts. As a case study, we use it to verify a small memory allocator that uses exceptions.

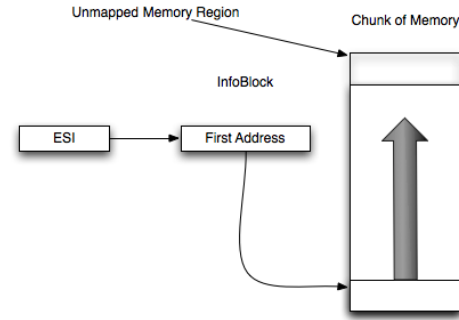
The source code to our mechanisation can be found at <http://www.itu.dk/people/mpav/downloads/exp-tgc05.zip>. The increment to the previous development amounts to 1084 lines of code. The code is compiled with `coqc` version 8.4p13 with OCaml 4.00.1.

## 2 Memory allocation using exceptions



We use the standard AT&T syntax for assembly notation. For this example, 'mov  $r$ ,  $v$ ' stores the value  $v$  in the register  $r$ , ' $[r]$ ' dereferences a pointer stored in  $r$  and 'add  $r$ ,  $v$ ' adds the value  $v$  to the value stored in the register  $r$ .

Jensen et al. [2] implemented and verified a simple bound-and-check memory allocator. We verify an alternative version, whose behaviour and code is depicted in Figure 1. In our allocator there are no checks for overflow or memory bounds, instead, we mark the end of the available memory with an *unmapped location*. The code takes an argument *info* that is a single pointer to the start of memory and begins by moving the starting address of the information block to the ESI register and then by moving its value to the EDI register. After this preamble, EDI will eventually point at the beginning of the available memory. Now we write the value 0 in the memory pointed by EDI. By writing a value to the byte of memory we wish to allocate we will trigger an exception if that memory is unmapped, i.e. when the end of the memory available to the allocator has been reached. It is then up to the interrupt handler to catch the exception, but by jumping to the *fail* address it will mimic the behaviour of the handler in previous work [2]. If the memory is mapped, the control flow will go through and add four bytes to the EDI register to keep track advance the pointer to the free memory. At this point we update the information block by storing the value of the EDI register into the value pointed by ESI.



```
alloc(info)  $\triangleq$  mov ESI, info;
                mov EDI, [ESI];
                mov [EDI], 0;
                add EDI, 4;
                mov [ESI], EDI.
```

Figure 1: Allocator code snippet

### 3 Allocator with exceptions specification

In order to give the specification of the piece of code in Figure 1 we use Jensen's step-indexed variant of separation logic, but here we prefer to keep the presentation as simple as possible, thus using standard separation logic connectives as  $\star$  for the usual *separating conjunction* in separation logic,  $\mapsto$  for a points-to predicate for the registers and  $\mapsto$  as a points-to predicate for the memory. Moreover, we use the question mark  $r?$  for registers and memory addresses as syntactic sugar for  $\exists, v. r \mapsto v$  and similarly for  $\mapsto$ . Here, we borrow the continuation passing style specification from previous work [2], thus, a specification has the following continuation-passing style form:

$$\vdash (\text{safe} \otimes Q \implies \text{safe} \otimes P) \circ i..j \mapsto c \quad (1)$$

which states that a program  $c$  stored in the memory from the address  $i$  and the address  $j$  is safe to run from  $P$  provided that there is a continuation that runs safely from  $Q$ .

The specification for the example in Figure 1 follows the same pattern, but, since the program can succeed or fault we need two continuations, one stating what happens upon success and one stating what happens upon failure, a pre-condition and a invariant (omitted for space reasons) specifying that there exists a storage which ends are bounded by an unmapped memory region and that there exists and IDT containing the pointers to the handlers.

We define the specification `allocSpec` as the pre and post-conditions of the code in Figure 1 stating that, when the code is pointed by the range of addresses  $i$  to  $j$ , is safe to execute from a state

$$P \triangleq \text{EIP} \mapsto i \star \text{INTL} \mapsto 0 \star \text{EDI?} \star \text{ESP} \mapsto sp \star (sp-4..sp) \mapsto spval \quad (2)$$

where `EIP` points to the beginning of the code, `INTL` is the register keeping track of the level of interruptions, `EDI` is a temporary register and `ESP` is the stack pointer, *provided* that the program is safe in case an exception occurs, i.e. that there exist an handler which is going to take on the computation from the address `fail` with the `INTL` set to 1 and the stack pointer containing the return address to the original code

$$Q_1 \triangleq \text{EIP} \mapsto fail \star \text{INTL} \mapsto 1 \star \text{EDI?} \star \text{ESP} \mapsto (sp-4) \star (sp-4..sp)? \quad (3)$$

and that there is a program which is safe run from the address  $j$  with the `EDI` register pointing to the end of the allocated memory and with the interrupt level set at zero in case the allocator succeeds

$$Q_2 \triangleq \text{EIP} \mapsto j \star \text{INTL} \mapsto 0 \star \text{ESP} \mapsto sp \star (sp-4)..sp \mapsto spval \star \\ \exists p, \text{EDI} \mapsto (p+4) \star (p..(p+4))? \quad (4)$$

By wrapping up the tree formulas all together we obtain the `allocSpec` specification:

$$\text{allocSpec} \triangleq \vdash ((\text{safe} \otimes Q_1 \wedge \text{safe} \otimes Q_2) \implies \text{safe} \otimes P) \otimes i..j \mapsto c \otimes \text{Inv}$$

Finally, we prove that implementation of the allocator respects the specification:

**Theorem 1.** *The specification `allocSpec` for the piece of code in Figure 1 is sound. [Coq proof]*

## 4 Conclusions and Future Work

We have extended an existing mechanisation of x86-assembly created by Jensen et al. to support synchronous interrupts. Jensen's model is expressive enough to reason about mutable code and we stay true to this design philosophy by storing the IDT and all handlers in memory, allowing them to be dynamically updated by the processor. Our extensions to the program logic are also very conservative. By allowing the memory points-to predicate to state that certain memory is unmapped (and not only what it contains), we obtain a logic that is expressive enough to verify programs that use synchronous interrupts. We believe that this is a testament not only to the validity of our design decisions, but also of the quality of the original mechanisation.

## References

- [1] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2 (2A, 2B & 2C): Instruction Set Reference*, January 2013.
- [2] J. B. Jensen, N. Benton, and A. J. Kennedy. High-level separation logic for low-level code. In *Proceedings of POPL*, pages 301–314. ACM, 2013.
- [3] Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. Coq: the world's best macro assembler? In *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 13–24, 2013.
- [4] The Coq Development Team. *The Coq Reference Manual, version 8.4*, 2012.

# Specialized Strategies for Learning Integrated Circuits using Angluin $L^*$ and Rivest/Shapire Homing Inference

Tanya Braun, Arne Wichmann, and Sibylle Schupp

Institute for Software Systems, Hamburg University of Technology, Hamburg, Germany  
 {tanya.braun, arne.wichmann, schupp}@tuhh.de

In a perfect world, every digital integrated circuit (IC) leads a well-documented life. Alas, documentations are lost, not written, deliberately withheld. To reconstruct the inner workings of an IC, we modify the learning algorithms by Dana Angluin and by Ronald Rivest and Robert Schapire so that they employ ICs as the learning environment. Known pin directions and functions allow reducing the state space, alphabet size, and set of learners and therefore accelerate the learning. We set up different strategies to seek out hidden state to realize an approximative equivalence check and provide the necessary counterexamples. In summary, the contributions are different strategies for the teacher to test for equivalence and a specialized learner for ICs, called **ALICE** [3], with the possibility to incorporate prior knowledge.

Angluin's  $L^*$  algorithm [1] learns an automaton representation using the inputs and the corresponding results plus the counterexamples obtained from a teacher, assuming we know a reset. Rivest and Schapire's algorithm [6] can be seen as a generalization of  $L^*$  and handles absent resets using homing sequences. Angluin has one learner whose queries are simulated from a unique state. To execute such a query in any environment, the environment must be reset to the unique state. In the absence of a reset, homing sequences bring the environment into a defined state. The learner, Angluin's main learning loop, becomes the basic building block. Rivest and Schapire's algorithm maintains a set of learners to accommodate the different states a homing sequence may lead to. For each final state the homing sequence can lead to, a learner exists that has this final state as a starting state.

**ALICE** uses the `libalf` library for the learners [2] and builds the homing algorithm around it. The IC takes the role of the teacher in the Angluin learning. **ALICE** incorporates prior knowledge about the ICs interface (see Table 1) into the model of the input alphabet and the learning process.

Prior Knowledge	Usage	Effect
Pin directions	Input = in + inout Output = out + inout	Reduced number of learners/ states, reduced alphabet size
Clock pin	Subtract from input alphabet, let hardware handle clock behaviour	Reduced alphabet size, reduced state space
Clear word	As homing sequence	One learner
Clear pin	Build homing sequence, subtract from input alphabet	One learner, reduced alphabet size

Table 1: Usage and Effects of Prior Knowledge

## 1 Strategies to Realize Equivalence Checks

The teacher has to provide a counterexample to an incorrect representation of an IC. An ideal check is impossible due to the infinite amount of stimulation necessary. We therefore

Strategy	Expected Effect
<b>0</b> No check, without any knowledge	Fewer queries than with any strategy, missing states with clocked ICs
<b>0c</b> No check, subtract clock pin	Fewer queries, fewer missing states with clocked ICs
<b>0d</b> No check, subtract clock and clear pin	Fewer queries than 0c, same state counts
<b>I</b> Walk to each state, check two times the input symbols; <b>base case</b>	Rather high number of learners, states, queries
<b>Ia</b> Base case strategy, clear word for reset (construct from pin, all zeros)	Only one learner (for those with a correct reset)
<b>Ib</b> Base case strategy, clear pin for reset, subtract it from input alphabet	Only one learner, fewer queries due to reduced alphabet
<b>Ic</b> Base case strategy, subtract clock pin (let teacher handle clock behaviour)	Reduced state space, fewer queries due to reduced alphabet/state space
<b>Id</b> Base case strategy, clear pin for reset, subtract clear and clock pin	Only one learner, reduced state space, fewer queries
<b>II</b> Walk to each state and toggle each pin	Fewer queries than with I, missing states
<b>III</b> Walk to each state; for each input, stimulate and toggle each pin	More queries and states than with II, performance similar to I
<b>IV</b> Walk to each state, stimulate ten random inputs, then toggle each pin	Fewer queries than with base case, more states than with II
<b>V</b> Block other learners and test with random inputs	Fewer queries than with base case

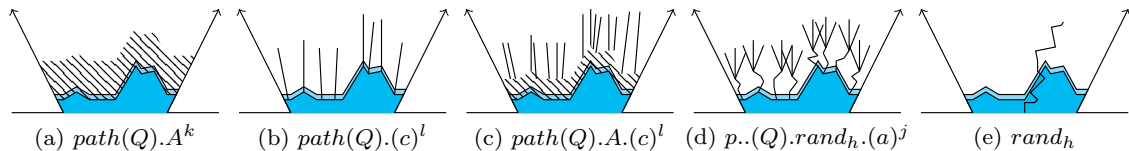
Table 2: Test Strategies for Evaluation

introduce different strategies, which are based on a general approximate equivalence check  $path(Q)?.A^k.rand_h.[(a)^j|(c)^l]$ , and list their expected effects (see Table 2).

The strategies **I** to **IV** use a  $path(Q)$  option to walk to the learned states and perform additional stimulation from the fringe of the explored state space. This part of the strategies avoids repeating questions that were already posed by the learner. In Fig. 1 these parts of the strategies are visualised as the coloured area.

The **0** strategies do not perform an equivalence check. Strategy **I** appends all possible combinations of two input alphabet elements and densely explores the area next to the learned area (see Fig. 1a). Strategy **II** tries to find first counterexamples fast using a toggle check plus path option (see Fig. 1b). A hybrid strategy (**III**) appends each symbol of the input alphabet and then toggles all pins (see Fig. 1c). The last path-based strategy (**IV**) appends 10 random symbols and then toggles all pins (see Fig. 1d).

A conceptually different strategy (**V**) does not systematically explore the state space, but instead performs a random walk using the input symbols to find a counterexample (see Fig. 1e). This strategy is close to the weak oracle introduced by Angluin and Rivest and Schapire.

Figure 1: The Search Space Covered by Equivalence Strategies<sup>1</sup>

<sup>1</sup> The x-axis represents the inputs as queries. The y-axis indicates the query length. The dark blue area represent the queries that lead to new states. The light blue area constitutes the input the learner looked further.

## 2 Evaluation

We evaluated the different equivalence strategies and applied levels of prior knowledge to several groups of ICs according to the number of queries, number of learners, and various performance parameters.

We use groups of ICs that stem from three university introductory courses to digital circuits [7, 4, 5]. The stateless groups include combinatoric logic gates like NAND with different numbers of input pins, de/encoders, (de)multiplexers, and arithmetic entities like comparators or full adders. The groups of ICs with state include shift registers, transceivers and buffers, flip-flops with clear pins or inverted outputs, and counters and oscillators. The evaluation shows that it is possible to learn the functionality of digital ICs.

Although ALICE needs no information about an IC except the number of pins as well as the power supply and ground pin, we can accelerate learning by nearly 100% by providing additional knowledge about pin directions and functions for selected groups.

Different equivalence checks vary considerably in their number of queries. The random-based strategies, often found in the literature, usually trigger a reset of the IC and terminate learning before an useful result is obtained (**IV**, **V**). On the other hand, for stateless ICs, an equivalence check is not necessary, and a **0** strategy with minimal cost is sufficient.

## References

- [1] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.
- [2] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The automata learning framework. In *Proceedings of the Twentysecond International Conference on Computer Aided Verification*, pages 360–364, 2010.
- [3] Tanya Braun. Teaching angluin to learn the inner workings of integrated circuits. Master’s thesis, Hamburg University of Technology, Hamburg, Germany, August 2015.
- [4] Anantha Chandrakasan. 6.374: Analysis and design of digital integrated circuits. Massachusetts Institute of Technology: MIT OpenCourseWare. Lecture Notes, 2003.
- [5] Anantha Chandrakasan. 6.111: Introductory digital systems laboratory. Massachusetts Institute of Technology: MIT OpenCourseWare. Lecture Notes, 2006.
- [6] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, pages 411–420, 1989.
- [7] Sven-Ole Voigt. Technische Informatik: Skript zur Vorlesung. Hamburg University of Technology. Lecture Notes, 2013.

# Binary session types for psi-calculi

Hans Hüttel

Department of Computer Science, Aalborg University, Denmark

## 1 Introduction

Binary session types arose in the work of Honda in the setting of the  $\pi$ -calculus; a binary session type describes the protocol followed by the two ends of a communication channel and a well-typed process will not exhibit communication errors, since the two ends of a channel must always adhere to the dual parts of their protocol. Binary session types have been used for describing a variety of program properties, including liveness properties. In the setting of sessions, an important property is that of progress, namely that a session will never be stuck waiting for a message that does not arrive.

This paper provides a common generalization of existing binary session type systems using psi-calculi. These have been proposed as a common framework for understanding the plethora of  $\pi$ -like process calculi; like  $\pi$ -like calculi psi-calculi have the notion of mobile names with scope but also allow channels to be not just names but arbitrary terms from a so-called nominal data type.

Type systems for psi-calculi already exist. In particular, there is a type system generalizing a collection of simple type systems and another system for resource-aware properties. We now extend this approach to a generic type system for binary session types. We only assume that session types have certain labelled transitions; this is in the tradition of behavioural contracts that provides a behavioural type discipline in which types have transitions.

A main result is the definition of a binary session type system for psi-calculi and a fidelity result that generalizes results from existing session type systems. Since channels can be arbitrary terms, a major challenge is to deal with this. Whenever a session is created, private session channels are introduced by means of scoped endpoint constructors that can be applied to ordinary terms in order to create a session channel. The type system keeps track of how the behaviour of a session channel evolves by keeping track of the modified behaviour of these endpoint constructors.

The safety result for our binary session type discipline is that of fidelity, namely that the usage of a well-typed channel evolves according to its session type.

Existing binary session type systems arise as instances of our general type system. These include a system for ensuring progress due to Vieira and Vasconcelos, a type system for correspondence assertions due to Vasconcelos et al. and the system with subtyping due to Gay and Hole.

## 2 Psi-calculi

Psi-calculus processes can contain *terms*  $M, N, \dots$ ; these must form a nominal datatype  $\mathbf{T}$ . If  $\Sigma$  is a signature, a nominal data type is then a  $\Sigma$ -algebra, whose carrier set is a nominal set. In the nominal data types of  $\psi$ -calculi we use simultaneous term substitution  $X[\tilde{z} := \tilde{Y}]$  which is to be read as stating that the terms in  $\tilde{Y}$  replace the names in  $\tilde{z}$  in  $X$ . We assume a notion of channel equivalence;  $\Psi \models M \leftrightarrow N$  denotes that terms  $M$  and  $N$  represent the same channel.

Processes can also contain *assertions*  $\Psi$  and *conditions*  $\varphi$ , that must also form nominal datatypes.

Unlike in the  $\pi$ -calculus channels can be arbitrary terms in a psi-calculus, as arbitrary terms are allowed in the subject position of a prefix.

We extend psi-calculus with the selection and branching primitives of Honda et al. , as these are standard in session calculi. In a selector was always a name; here we allow arbitrary terms  $M$  as selectors. Branching thus becomes  $M \triangleright \{l_1 : P_1, \dots, l_k : P_k\}$  and selection is written as  $M \triangleleft l.P_1$ , where  $l$  ranges over a set of label names.

We introduce session channels by means of dual endpoints as in Giunti and Vasconcelos . The construct  $(\nu c)P$  can be used to set up a new session channel with endpoint constructor  $c$  that can be used to build session channels.

All in all, this gives us the formation rules

$$P ::= \underline{M}(\lambda \tilde{x})X.P \mid \overline{MN}.P \mid P_1 \mid P_2 \mid (\nu c)P \mid !P \mid (\Psi) \mid \mathbf{case} \varphi_1 : P_1, \dots, \varphi_k : P_k \\ \mid M \triangleleft l.P_1 \mid M \triangleright \{l_1 : P_1, \dots, l_k : P_k\}$$

### 3 A generic type system

We  $T$  range over the set of types and distinguish between *base types*  $B$ , *session types*  $S$  and *endpoint types*  $T_E$ . An endpoint type  $T_E$  describes the behaviour at one end of a channel. A session type  $S$  describes the behaviour at both ends of a channel and is an unordered pair  $(T_1, T_2)$  of endpoint types, i.e. so  $(T_1, T_2)$  and  $(T_2, T_1)$  denote the same type.

In psi-calculi channels can be arbitrary terms; in our setting we use *session constructors* to indicate that a term is to be used as a session channel. A term whose principal session constructor is  $c$  will have a type of the form  $T@c$ .

We assume a deterministic labelled transition relation defined on the set of endpoint types. Transitions are of the form  $T_E \xrightarrow{\lambda} T'$  where

$$\lambda ::= !T_1 \mid ?T_1 \mid \triangleleft l \mid \triangleright l$$

If a channel has endpoint type  $T_E$ , which has the transition  $T_E \xrightarrow{?T_1} T'_E$ , then following an input of a term of type  $T_1$ , the channel will now have endpoint type  $T'_E$ . For a given type language, we must give transition rules that describe how these transitions arise.

We assume a *duality condition* for labels in labelled type transitions; we define  $\overline{!T_1} = ?T_2$  and  $\overline{\triangleleft l} = \triangleright l$  and vice versa, and we require that  $\overline{\overline{\lambda}} = \lambda$ . A session type is balanced if the types of its endpoint are dual to each other.

**Definition 1.** A session type  $S$  is balanced if  $S = (T_E, \overline{T_E})$  for some  $T_E$ .

A type environment  $\Gamma$  is a finite function from names to types, often written as  $\tilde{x} : \tilde{T}$ . A type environment  $\Gamma$  is balanced if for every  $x \in \text{dom}(\Gamma)$  we have that whenever  $\Gamma(x) = S$ , then  $S$  is balanced.

**Type judgements** The type judgements in our type system are of the form  $\Gamma, \Psi \vdash \mathcal{J}$  where  $\mathcal{J}$  is built using the formation rules

$$\mathcal{J} ::= M : T \mid X : \tilde{T} \rightarrow U \mid \Psi \mid \varphi \mid P$$

For terms, the type judgment  $\Gamma, \Psi \vdash M : T@c$  says that the term  $M$  has type  $T$  using session constructor  $c$ . The rules defining these judgements depend on the instance of the type system but we require that the session constructor must have an endpoint type for the resulting channel to be typable. Rules for assertions and conditions are also specific to the instance considered.

The type rules for *processes* contain judgment of the form  $\Gamma, \Psi \vdash P$  where  $\Psi$  is an assertion. Table 1 contains the most interesting type rules for processes.

Note that for patterns, judgments are of the form  $\Gamma, \Psi \vdash X : \tilde{T} \rightarrow U$ . The intended interpretation is that pattern  $X$  has type  $\tilde{T} \rightarrow U$  if the pattern variables are bound to terms of types  $\tilde{T}$  whenever the pattern matches a term of type  $U$ .

An important rule is that for parallel composition, since type addition enables us to split a session type into two endpoint types; this follows Giunti and Vasconcelos .

Whenever a prefix is typed, the type of the subject must be updated when typing the continuation. As subjects in the psi-calculus setting can be arbitrary terms, we update the type of the channel constructor used to construct the channel.

(OUTPUT)	$\frac{\Gamma_1, \Psi_1 \vdash_{\min} M : T_1@c \quad T_1 \xrightarrow{!,T_2} T_3 \quad \Gamma_2, \Psi_2 \vdash_{\min} N : T_2 \quad \Gamma_3 + c : T_3, \Psi_3 \vdash P}{\Gamma_1 + \Gamma_2 + \Gamma_3, \Psi_1 \otimes \Psi_2 \otimes \Psi_3 \vdash \overline{MN}.P}$		
(INPUT)	$\frac{\Gamma_1, \Psi_1 \vdash_{\min} M : T_1@c \quad T_1 \xrightarrow{?,T_2} T_3(\tilde{x}) \quad \Gamma_2, \Psi_2 \vdash_{\min} X : \tilde{U} \rightarrow T_2 \quad \Gamma_3 + \tilde{x} : \tilde{U} + c : T_3[\tilde{x}], \Psi_3 \vdash P}{\Gamma_1 + \Gamma_2 + \Gamma_3, \Psi_1 \otimes \Psi_2 \otimes \Psi_3 \vdash \underline{M}(\lambda\tilde{x})X.P}$	$\tilde{x} \# \text{dom}(\Gamma_1 + \Gamma_2 + \Gamma_3)$ $\tilde{x} \# \Psi_1 \otimes \Psi_2 \otimes \Psi_3$	
(CASE)	$\frac{\Gamma, \Psi \vdash \varphi_i \quad \Gamma, \Psi \vdash P_i \quad 1 \leq i \leq k}{\Gamma, \Psi \vdash \mathbf{case} \varphi_1 : P_1, \dots, \varphi_k : P_k}$	(SESSION)	$\frac{\Gamma + x : T, \Psi \vdash P}{\Gamma, \Psi \vdash (\nu x : T)P} \quad x \# \Gamma, \Psi$
(SELECT)	$\frac{\Gamma_1, \Psi_1 \vdash_{\min} M : T@c \quad \Gamma_2 + c : T_i, \Psi_2 \vdash P \quad T \xrightarrow{\triangleleft, l_i} T_i}{\Gamma_1 + \Gamma_2, \Psi_1 \otimes \Psi_2 \vdash M \triangleleft l_i.P}$		
(BRANCH)	$\frac{\Gamma_1, \Psi_1 \vdash_{\min} M : T@c \quad T \xrightarrow{\triangleright, l_i} T_i \text{ and } \Gamma_2 + c : T_i, \Psi_{2i} \vdash P_i \text{ for } 1 \leq i \leq k}{\Gamma_1 + \Gamma_2, \Psi_1 \otimes \bigotimes_{i=1}^k \Psi_{2i} \vdash M \triangleright \{l_1 : P_1, \dots, l_k : P_k\}}$		

Table 1: Selected type rules

**Theorem 1.** *Suppose we have  $\Psi_0 \blacktriangleright P \xrightarrow{\alpha} P'$ , where  $\alpha$  is a  $\tau$ -action and that  $\Gamma, \Psi \vdash_{\text{bal}} P$  and  $\Psi \leq \Psi_0$ . Then for some  $\Psi' \leq \Psi$  and  $\Gamma' \leq \Gamma$  we have  $\Gamma' \vdash_{\min} \alpha : (T@c, U)$ .*

**Theorem 2 (Fidelity).** *Suppose we have  $\Psi_0 \blacktriangleright P \xrightarrow{\alpha} P'$ , where  $\alpha$  is a  $\tau$ -action and that  $\Gamma, \Psi \vdash P$  with  $\Gamma$  and  $\Gamma_P$  balanced and  $\Psi \leq \Psi_0$ . Then for some  $\Psi' \leq \Psi$  we have  $\Gamma \uparrow \alpha, \Psi' \vdash_{\text{bal}} P'$ .*



# Towards A Hybrid Approach to Software Verification (Extended Abstract)

Dario Della Monica<sup>1</sup> and Adrian Francalanza<sup>2</sup>

<sup>1</sup> ICE-TCS, Reykjavik University, Iceland  
dariodm@ru.is

<sup>2</sup> CS, ICT, University of Malta, Malta  
adrian.francalanza@um.edu.mt

*Model checking* (MC) [6] is a widely accepted *pre-deployment* verification technique that checks whether a system satisfies or violates a property by potentially analysing *all* the possible system behaviours. By contrast, *runtime verification* (RV) [10, 14] is a lightweight verification technique aimed at mitigating scalability issues such as state explosion problems, typically associated with traditional verification techniques like MC. RV attempts to infer the satisfaction (or violation) of a correctness property from the analysis of the *current execution* of the system under scrutiny. It is thus performed *post-deployment* (on actual system execution), which is appealing for component-based applications (parts of which may not be available for analysis pre-deployment), as well as for dynamic settings such as mobile computing (where components are downloaded and installed at runtime). The technique has fostered a number of verification tools, e.g., [2, 3, 8, 9, 12, 13, 16], and has proved effective in various scenarios [4, 7, 17].

Despite its advantages, RV is limited when compared to MC because certain correctness properties cannot be verified at runtime [5, 10, 15]. For instance, MC makes it possible to check for both *safety* and *liveness* properties, by providing either a positive or a negative answer, according to whether the system conforms with the specifications; RV, on the other hand, can only return a positive verdict for certain liveness properties (called co-safety properties [5]) or a negative one for safety conditions. Moreover, RV induces a runtime overhead over the execution of a monitored system, which should ideally be kept to a minimum [14].

RV's limits in terms of verifiable properties is evidenced more for branching-time logics, that are able to express properties describing behaviour over multiple system executions. In recent work [11], one such branching-time logic called  $\mu$ HML [1] is studied from an RV perspective. Figure 1 outlines the logic  $\mu$ HML used and its semantics, defined over a *Labelled Transition System* (LTS), consisting of a set of *states*  $s, r \in \text{STA}$ , sets of *actions*  $\alpha \in \text{ACT}$ , and a *transition relation* between states labelled by actions,  $s \xrightarrow{\alpha} r$ ; as in [1], the semantic definition employs an *environment* from  $\mu$ HML logical variables,  $\text{VARS}$ , to sets of states,  $\rho \in (\text{VARS} \rightarrow \mathcal{P}(\text{STA}))$ . One of the main contributions of [11] is the identification of an *expressively maximal, runtime-verifiable subset* of the logic, reported in Figure 1 as the grammar for SHML and CHML; in [11] they show how these classes provide an easy syntactic check for determining whether a property satisfaction (or violation) can be determined using the RV technique.

We building on the findings of [11], with the aim of extending the applicability of RV to a larger class of  $\mu$ HML properties other than  $\text{SHML} \cup \text{CHML}$  from Figure 1. Specifically, we propose a *hybrid approach* that permits automated formal verification to be spread across the pre- and post-deployment phases of a system development, with the aim of calibrating the management of the verification burden while combining the strengths of MC with those of RV. As an illustrative example, consider the  $\mu$ HML property (1) below, describing systems that *can* perform action  $a$ , prefix  $\langle a \rangle(\dots)$ , and reach a state from where it *can either* perform action

**Syntax**

$\varphi, \phi \in \mu\text{HML} ::= \text{tt}$	(truth)	$\text{ff}$	(falsehood)
$\varphi \vee \phi$	(disjunction)	$\varphi \wedge \phi$	(conjunction)
$\langle \alpha \rangle \varphi$	(possibility)	$[\alpha] \varphi$	(necessity)
$\min X. \varphi$	(min. fixpoint)	$\max X. \varphi$	(max. fixpoint)
$X$	(rec. variable)		

**Semantics**

$\llbracket \text{tt} \rrbracket$	$\stackrel{\text{def}}{=} \text{STA}$	$\llbracket \text{ff} \rrbracket$	$\stackrel{\text{def}}{=} \emptyset$
$\llbracket \varphi_1 \vee \varphi_2 \rrbracket$	$\stackrel{\text{def}}{=} \llbracket \varphi_1 \rrbracket \cup \llbracket \varphi_2 \rrbracket$	$\llbracket \varphi_1 \wedge \varphi_2 \rrbracket$	$\stackrel{\text{def}}{=} \llbracket \varphi_1 \rrbracket \cap \llbracket \varphi_2 \rrbracket$
$\llbracket \langle \alpha \rangle \varphi \rrbracket$	$\stackrel{\text{def}}{=} \{s \mid \exists r. s \xrightarrow{\alpha} r \text{ and } r \in \llbracket \varphi, \rho \rrbracket\}$	$\llbracket [\alpha] \varphi \rrbracket$	$\stackrel{\text{def}}{=} \{s \mid \forall r. s \xrightarrow{\alpha} r \text{ implies } r \in \llbracket \varphi, \rho \rrbracket\}$
$\llbracket \min X. \varphi \rrbracket$	$\stackrel{\text{def}}{=} \bigcap \{S \in \text{STA} \mid \llbracket \varphi, \rho[X \mapsto S] \rrbracket \subseteq S\}$	$\llbracket \max X. \varphi \rrbracket$	$\stackrel{\text{def}}{=} \bigcup \{S \in \text{STA} \mid S \subseteq \llbracket \varphi, \rho[X \mapsto S] \rrbracket\}$
$\llbracket X \rrbracket$	$\stackrel{\text{def}}{=} \rho(X)$		

**Monitorable Fragments**

$\theta, \vartheta \in \text{sHML} ::= \text{tt}$	$\text{ff}$	$[\alpha] \theta$	$\theta \wedge \vartheta$	$\max X. \theta$	$X$
$\pi, \varpi \in \text{cHML} ::= \text{tt}$	$\text{ff}$	$\langle \alpha \rangle \pi$	$\pi \vee \varpi$	$\min X. \pi$	$X$

Figure 1:  $\mu\text{HML}$  Syntax and Semantics

$b$ , subformula  $\langle b \rangle \text{tt}$ , or else *can never* perform action  $c$ , subformula  $[c] \text{ff}$ .

$$\langle a \rangle (\langle b \rangle \text{tt} \vee [c] \text{ff}) \quad (1)$$

According to Figure 1, (1) turns out *not* to be runtime-verifiable because of the subformula  $[c] \text{ff}$ ; intuitively, whereas a system execution exhibiting action  $a$  followed by action  $b$  suffices to prove that the system satisfies (1), an RV monitor cannot determine whether a system can never produce action  $c$  after performing action  $a$  from the observation of only a *single* system execution [11]. However, property (1) can be expressed as the (logically equivalent) formula

$$(\langle a \rangle \langle b \rangle \text{tt}) \vee (\langle a \rangle [c] \text{ff}) \quad (2)$$

whereby we note that the subformula  $\langle a \rangle \langle b \rangle \text{tt}$  is *runtime verifiable*, according to [11]. We argue that reformulations such as (2) allow for a *hybrid* compositional approach to verification, where part of the property, e.g., the subformula  $\langle a \rangle [c] \text{ff}$ , can be checked prior system deployment using MC, and the remaining part of the property, e.g.,  $\langle a \rangle \langle b \rangle \text{tt}$ , can be runtime-verified during system execution.

Preliminary investigations indicate that this decomposition approach applies to arbitrary  $\mu\text{HML}$  formulas. We therefore aim to devise *general* analysis techniques that reformulate *any*  $\mu\text{HML}$  formula into either conjunctions or disjunctions, i.e.,  $\varphi_{\text{RV}} \wedge \varphi_{\text{MC}}$  or  $\varphi_{\text{RV}} \vee \varphi_{\text{MC}}$ , where  $\varphi_{\text{RV}}$  and  $\varphi_{\text{MC}}$  denote the runtime-verifiable and model-checkable formula components, respectively. From a software engineering perspective, we envisage at least two ways how this decomposition between pre- and post-deployment verification can be fruitful:

1. The ensuing hybrid approach may be used as a means to *minimise* the verification effort required *prior to the deployment* of a system. E.g., in the case of (2), the model-checked subformula  $\varphi_{\text{MC}} = \langle a \rangle [c] \text{ff}$  is *smaller* than the full formula (1), since we would be offloading a degree of verification onto the runtime phase when runtime-verifying for

$\varphi_{RV} = \langle a \rangle \langle b \rangle \text{tt}$ . Moreover, for disjunction decompositions such as (2), the satisfaction of  $\varphi_{MC}$  prior to deployment obviates the need for any runtime analysis, minimising runtime overheads (a dual argument applies for conjunction decompositions and  $\varphi_{MC}$  violations).

2. In settings where software correctness is desirable but not essential, a hybrid approach can be used as a means to circumvent full-blown MC. Specifically, instead of model-checking for (1), a system may be runtime-verified for  $\varphi_{RV} = \langle a \rangle \langle b \rangle \text{tt}$  during its pilot launch, acting as a *vetting phase*: if  $\varphi_{RV}$  is satisfied during RV, this means that, by (2), (1) is satisfied as well; if not, we then proceed to model-check the system offline *wrt.*  $\varphi_{MC} = \langle a \rangle [c] \text{ff}$ .

## References

- [1] L. Aceto, A. Ingólfssdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge Univ. Press, 2007.
- [2] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In D. Giannakopoulou and D. Mry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012.
- [3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 44–57. Springer Berlin Heidelberg, 2004.
- [4] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on Martian rover software. *FMSD*, 25(2-3):167–198, 2004.
- [5] E. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *ALP*, volume 623 of *LNCS*, pages 474–486. Springer-Verlag, 1992.
- [6] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [7] C. Colombo and G. J. Pace. Fast-forward runtime monitoring - an industrial case study. In *Runtime Verification*, volume 7687 of *LNCS*, pages 214–228. Springer, 2012.
- [8] B. D’Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. In *TIME*, pages 166–174. IEEE, June 2005.
- [9] N. Decker, M. Leucker, and D. Thoma. jUnitRV - Adding Runtime Verification to jUnit. In *NASA Formal Methods*, volume 7871 of *LNCS*, pages 459–464. Springer, 2013.
- [10] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.
- [11] A. Francalanza, L. Aceto, and A. Ingólfssdóttir. On verifying Hennessy-Milner logic with recursion at runtime. In *Runtime Verification*. Springer, 2015. (to appear).
- [12] A. Francalanza and A. Seychell. Synthesising Correct Concurrent Runtime Monitors. *FMSD*, pages 1–36, 2014.
- [13] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *FMSD*, 24(2):129–155, 2004.
- [14] M. Leucker and C. Schallhart. A brief account of Runtime Verification. *JLAP*, 78(5):293–303, 2009.
- [15] Z. Manna and A. Pnueli. Completing the Temporal Picture. *TCS*, 83(1):97–130, 1991.
- [16] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.
- [17] S. Varvaressos, D. Vaillancourt, S. Gaboury, A. Blondin Mass, and S. Hall. Runtime monitoring of temporal logic properties in a platform game. In *Runtime Verification*, volume 8174 of *LNCS*, pages 346–351. Springer, 2013.

# Compiling Protocol Narrations into Applied Pi Processes

Hans Hüttel and Sam Sepstrup Olesen

Department of Computer Science, Aalborg University, Denmark

When designing or presenting cryptographic protocols, it is common to use informal protocol descriptions, such as *protocol narrations*, to describe the intended execution of a protocol run as a sequence or directed graph of message exchanges between the associated principals[BN07]. An example of a protocol narration can be seen below. Encryption of a message  $M$  with a key  $N$  denoted by  $\{M\}_N$ .

$$\begin{aligned} A &\rightsquigarrow S : \{B\}_{k_{AS}} \\ S &\rightsquigarrow A : \{k_B\}_{k_{AS}} \\ A &\rightsquigarrow B : \{m\}_{k_B} \end{aligned}$$

This narration describes a simple protocol, in which a principal  $A$  can ask a key server  $S$  for a key to communicate with a principal  $B$ . However, some assumptions about ownership of keys are made, namely that  $A$  and  $S$  share a key  $k_{AS}$  and that  $B$  knows the key  $k_B$  that is sent to  $A$  by  $S$ . Other important aspects are the actions that principals are supposed to perform on messages that are received, e.g. decryption and consistency checks are implicit. We obviously expect the principals to attempt decryption of the messages they receive. We also expect that the key-server  $S$  only sends the key  $k_B$ , if that is the key requested by  $A$ .

An important challenge is to be able to create an executable implementation of the protocol based on a narration. In [BN07] Briais and Nestmann define a translation of protocol narrations to processes in the spi calculus, which have been implemented by Briais in the `spyer` compiler[Bri08]. However, the spi calculus contains a fixed set of cryptographic primitives, and protocols that depend on other cryptographic primitive, e.g. homomorphic encryption, are therefore not supported by the compiler.

In this paper we deal with this issue by instead considering the applied pi calculus[AF01] which is an extension to the spi calculus with arbitrary cryptographic primitives. These primitives are described by means of an equational theory. For instance, the equation  $\text{dec}(\text{enc}(x, y), y) = x$  describes how symmetric encryption and decryption of a message  $x$  with a key  $y$  should behave.

We describe how to translate protocol narrations into a version of the applied pi calculus; this method is implemented in OCaml as a tool which acts as a front-end for ProVerif.

In addition to the translation from protocol narrations to processes in the spi calculus[BN07], our work is related to that of projection from global session types to local session types[HYC08] and code generation based on protocol narrations[Mod14].

While the addition of an equational theory for deriving terms is useful for modelling arbitrary cryptographic primitives, they can be the source of high computational complexity. We show that the problem of deriving terms using an equational theory and a set of terms is NP-complete.

**Definition 1** (Term Derivation Problem). *Given a signature  $\Sigma$ , an equational theory  $\Theta$ , a set of terms  $T$ , and a term  $t$ , can a term be synthesised from  $\Sigma$  and  $T$  that equals  $t$  in  $\Theta$ ?*

**Theorem 1.** *The term derivation problem is NP-Complete.*

easychair: Running title head is undefined.

easychair: Running author head is undefined.

## References

- [AF01] Martín Abadi and Cédric Fournet. “Mobile values, new names, and secure communication”. In: *ACM SIGPLAN Notices* 36.3 (2001), pp. 104–115.
- [BN07] Sébastien Briaïs and Uwe Nestmann. “A formal semantics for protocol narrations”. In: *Theoretical Computer Science* 389.3 (2007), pp. 484–511.
- [Bri08] Sébastien Briaïs. *spyer user’s guide*. 2008. URL: <http://sbriaïs.free.fr/tools/spyer/>.
- [HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. “Multiparty asynchronous session types”. In: *ACM SIGPLAN Notices* 43.1 (2008), pp. 273–284.
- [Mod14] Paolo Modesti. “Efficient Java Code Generation of Security Protocols Specified in AnB/AnBx”. In: *Security and Trust Management*. Vol. 8743. Springer, 2014, pp. 204–208.

# A generalization of termination conditions for partial model completion

Fazle Rabbi<sup>ab</sup>, Yngve Lamo<sup>a</sup>, Lars Michael Kristensen<sup>a</sup>, and Ingrid Chieh Yu<sup>b</sup>

<sup>a</sup> Bergen University College, Bergen, Norway

<sup>b</sup> University Of Oslo, Oslo, Norway

fra@hib.no, Yngve.Lamo@hib.no, lmkr@hib.no, ingridcy@ifi.uio.no

## 1 Introduction

It is ironic that Model Driven Engineering (MDE) was introduced to reduce the complexity of system development, but in many cases, adds accidental complexity [7]. In the process of development, software designers are often confronted with a variety of inconsistencies and/or incompleteness in the models under construction [4]. In particular, the modeller will most of the time be working with a *partial model* not conforming (i.e., being typed by and satisfying modelling constraints) to the metamodel that defines the modelling language being used [6]. Clearly, the productivity of the modeller could be improved by providing editing support that could either automatically fix a partial model or make suggestions based on *completion rules* to assist the modeller in completing the model [5]. In many respects, this idea is similar to code completion features as found in IDEs. More generally, complexity of modelling could be reduced by providing editing support for automated rewriting of models so that they conform to the modelling language used. However the philosophy of this approach incorporates an important element: in the form of ‘termination analysis’. In order to guarantee termination of the model completion, we propose a set of sufficient termination criteria.

### 1.1 Example of a model completion

In [5] we presented a web-based metamodeling and transformation tool called WebDPF based on the Diagram Predicate Framework (DPF) [2] which supports multilevel metamodeling and allows partial model completion. DPF provides an abstract visualization of concrete constraints. In WebDPF, one can graphically specify completion rules. WebDPF exploits the locality of model transformation rules and provides a foundation that enable automated tool-support to increase modelling productivity. The WebDPF editor (see Figure 1) consists of four resizable windows. The windows are arranged in a single view which provides the modeller with an overview of different modelling artefacts. The control panel on the left allows the user to select metamodels from the metamodel stack and also provides options to perform analysis such as conformance checking and termination. The conformance checking is used for validating whether a model conforms to its metamodel and the termination analysis is used for checking whether the application of transformation rules are terminating. While designing a model using the WebDPF model editor, the metamodel viewer displays the metamodel to help the modeller choose types for modelling elements. The signature editor is used to graphically define the arity and visualization of predicates. An atomic constraint can be defined by selecting a predicate in the signature editor and binding the arity of the predicate to the model elements from the model editor. Figure 1 shows a predicate called *composition* (*[comp]*) in the signature editor, and its associated completion rule in the completion rule editor. The purpose of the completion rule is to fix a model with missing edges. A partial model instance is shown in Figure 2 which

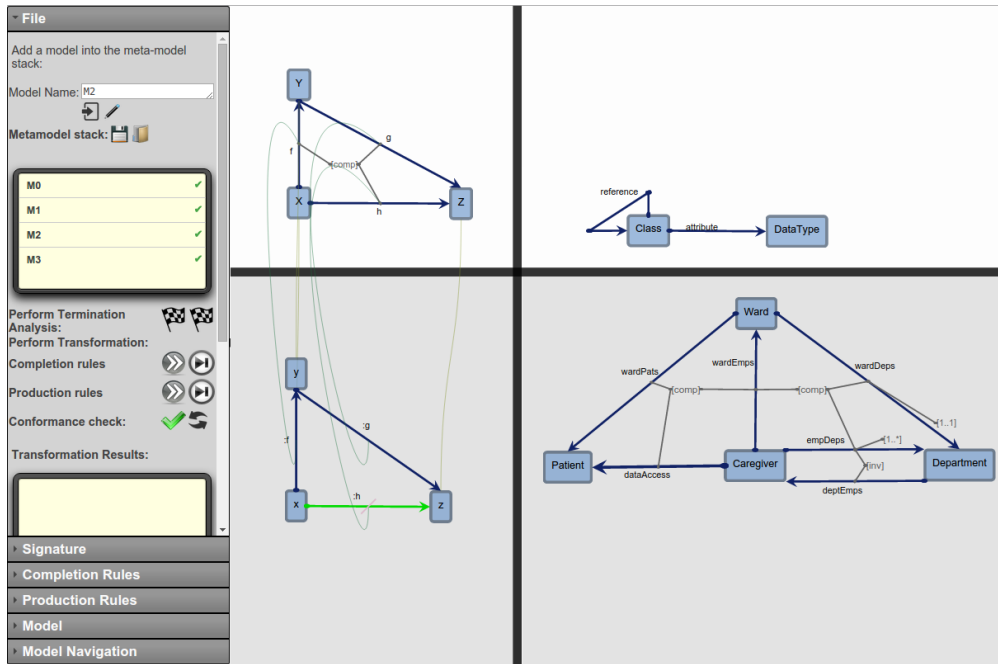


Figure 1: The WebDPF editor with a control panel (left), a metamodel viewer (top right), a model editor (bottom right), a signature editor (top middle), and a completion rule editor (bottom middle)

does not satisfy all the atomic constraints. After applying completion rules, the partial model instance becomes a complete model instance.

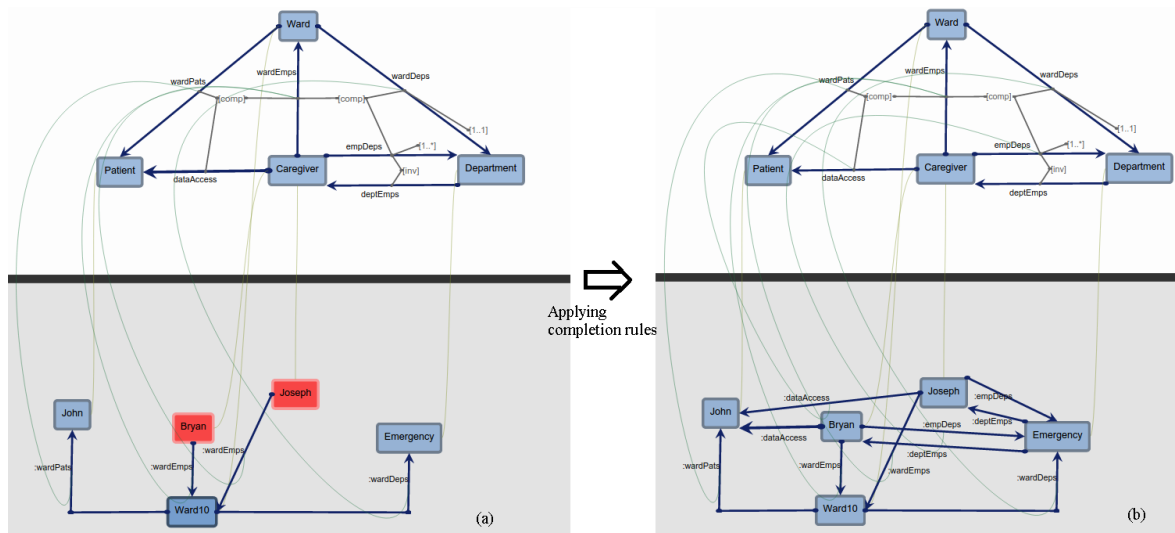


Figure 2: (a) An inconsistent model instance and (b) a complete model instance

## 2 Termination Criterion

In this article we focus on termination analysis for the application of transformation rules for model completion. Our proposed analysis is based on the principles adapted from layered graph grammars [1]. In a layered typed graph grammar, transformation rules are distributed across different layers. The transformation rules of a layer are applied as long as possible before going to the next layer. Ehrig et al [1] distinguished between deletion and nondeletion layers where all transformation rules in deletion layers delete at least one element and all transformation rules in nondeletion layers do not delete any elements, but the rules have negative application conditions. A set of layer conditions was specified in [1] that need to be satisfied by each layer  $k$  to guarantee termination. The layer conditions in [1] do not permit a rule  $r$  to use an element  $x$  of a given type  $t$  for the match if any element of type  $t$  has been created in a layer  $k' \leq k$ . The layer conditions also imply that the creation layer of an element of type  $t$  must precede its deletion layer. This restriction prevents the repetitive application of a certain rule. This layered typed graph grammar approach is suitable for situations where repetitive application of rules are not required. Unfortunately, there are many situations where repetitive application of rules are desirable such as to compute transitive closure operations [3].

To overcome the limitations of [1], discussed above, we generalize the layer conditions from [1] allowing deleting and non-deleting rules to reside in the same layer as long as the rules are loop-free. Furthermore, our generalization permits a rule to use newly created edges allowing us to perform transitive closure operations. A loop detection algorithm is implemented that overestimates the existence of a loop from a set of rules. Let  $R_k$  be the set of rules of a layer  $k$ . Our loop detection algorithm is based on the following sufficient conditions for loop freeness.

- *Cond1*: If a rule  $r_i \in R_k$  creates an element  $x$  of type  $t$ , then  $r_i$  must have  $x$  in its negative application condition,
- *Cond2*: If a rule  $r_i \in R_k$  deletes an element of type  $t$ , then there is no rule in  $r_j \in R_k$  that creates an element of type  $t$ ,

Note that we are assuming, that there are a finite number of rules in each layer and that the rules are applied on a finite input graph. The algorithm we developed guarantees termination if all the rules for each layer satisfies the above mentioned conditions. We wish to cover the following topics during our presentation:

- Generalized termination conditions
- Proof of correctness of our algorithm
- Discussion on complexity of the algorithm

## References

- [1] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006.
- [2] Y. Lamo, X. Wang, F. Mantz, W. MacCaull, and A. Rutle. Dpf workbench: A diagrammatic multi-layer domain specific (meta-)modelling environment. In R. Lee, editor, *Computer and Information Science 2012*, volume 429 of *Studies in Computational Intelligence*, pages 37–52. Springer, 2012.
- [3] T. Levendovszky, U. Prange, and H. Ehrig. Termination criteria for dpo transformations with injective matches. *Electr. Notes Theor. Comput. Sci.*, 175(4):87–100, 2007.



- 
- [4] T. Mens and R. Van Der Straeten. Incremental resolution of model inconsistencies. In *Recent Trends in Algebraic Development Techniques*, volume 4409 of *LNCS*, pages 111–126. Springer, 2007.
  - [5] F. Rabbi, Y. Lamo, I. C. Yu, and L. M. Kristensen. A diagrammatic approach to model completion. In *4th Workshop on the Analysis of Model Transformations (AMT) @ MODELS'15*, *accepted*, 2015.
  - [6] S. Sen, B. Baudry, and D. Precup. Partial model completion in model driven engineering using constraint logic programming. In *INAP'07*, Germany, 2007.
  - [7] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *16th International Conference, MODELS 2013*, volume 8107 of *LNCS*, pages 1–17. Springer, 2013.

# Flooding Detection in Concurrent Object Systems\*

Charlie McDowell<sup>†</sup>  
University of California, Santa Cruz, USA  
mcdowell@ucsc.edu

Olaf Owe<sup>‡</sup>  
University of Oslo, Norway  
olaf@ifi.uio.no

## Abstract

Concurrency is today an essential component of computer systems. One approach to programming concurrent object oriented systems is the use of active objects and asynchronous method calls, based on the actor model. This model is attractive by offering efficient programming and a simple, compositional semantics. The model facilitates independent units with a high degree of concurrency, but may also lead to deadlock. In this paper we show that systems developed using active objects and asynchronous method calls can result in system failure due to over-eager concurrency, which we call *flooding*. If an concurrent unit is flooded it is not able to respond properly. We present an algorithm to statically detect flooding, and we prove the soundness of the algorithm.

**Keywords:** active objects, concurrent objects, asynchronous communication, futures, static analysis, concurrency faults

## 1 Introduction

Concurrency is today a key aspect of the computer systems forming our infra-structure. This aspect is essential in distributed systems and net-based service systems such as cloud computing, as well as multi-core computers. Since it is easier to reduce parallelism than to increase the amount of parallelism, it is a non-trivial challenge to design systems that allow the desired amount of concurrency – and in a correct manner. In practice many systems rely on centralized control or synchronization of blocks of code to make programs dealing with shared data work correctly, including thread-based object-oriented concurrency, which is the most common paradigm used to program distributed systems today. However, synchronization restricts parallelism and slows down overall performance. While synchronization primitives for notification/signaling may improve efficiency, they are difficult to use correctly because they break modular reasoning and understanding.

The Actor model has been acknowledged as a natural way of programming concurrent systems, and is based on a simple semantics allowing modular reasoning [7, 2, 1]. It has been extended to the object-oriented setting in the form of active concurrent objects, interacting by means of remote method calls. Asynchronous methods increase efficiency by allowing non-blocking calls [9, 8]; and shared futures enable even more efficient interaction, allowing objects to share computation results without waiting for the results [13, 6, 10, 12, 4]. For instance a caller who does not need the result of the called method may pass the future identity of the result to other objects without (itself) waiting for the result to appear.

We consider a high-level core language based on this concurrency model. The language includes a mechanism for asynchronous call, suspension of the active process, and blocking and non-blocking primitives for obtaining future values, similar to [5]. Inter-object concurrency comes for free in the sense that each object can run concurrently with other objects. Intra-object synchronization is handled in a modular manner without the use of external notification. The concurrency model allows unrestricted

---

\*This work was done in the context of the EU projects FP7-610582 *Envisage: Engineering Virtualized Services* (<http://www.envisage-project.eu>) and FP7-ICT-2013-X *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations* (<http://www.upscale-project.eu>).

<sup>†</sup>

<sup>‡</sup>

concurrency with a compositional semantics. Thus it enables efficient programming, class-wise understanding, verification, and testing. However, this unrestricted concurrency model does not come without a price. This programming style may give rise to deadlocks, and it is easy to create programs that are class-wise semantically correct but that fail due to over-eager creation of method calls. A system may feed an object with more calls than it is able to handle, regardless of its processing speed. We refer to this situation as *flooding* of the object.

In this paper we define and exemplify the concept of flooding, distinguishing between strong and weak flooding. Weak flooding is less serious than strong flooding and can be managed with the use of fair scheduling of the processes within an object. Strong flooding may eventually overwhelm a system, even in the presence of fair scheduling. The scientific contribution of the paper is to propose a static analysis method to detect possible flooding situations, and prove its soundness (no false negatives). Since static analysis of flooding cannot be both sound and complete, detection of flooding may not imply a real flooding situation. However, when no flooding is detected, this implies that there is no real flooding situation (soundness).

While analysis of deadlock situations for this concurrency model has been investigated in several ways, we are not aware of analysis of object flooding for this concurrency model. Arvind and Nikhil [3] recognized a problem of “excessive parallelism” in the context of the functional dataflow language Id and tagged-token dataflow. More recently, there have been efforts to address scheduling and fairness issues with active objects, but none of that work discusses the issue of system failure due to flooding. Instead, scheduling has been proposed to improve performance, and in some cases as an essential part of the correctness of the algorithm [11].

## References

- [1] G. Agha, S. Frolund, W.Y. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(2):3–14, may 1993.
- [2] Gul A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.
- [3] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. on Computers*, 39(3):300–318, March 1990.
- [4] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer-Verlag, 2004.
- [5] Crystal C. Din and Olaf Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27:1–22, 2014.
- [6] Robert H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
- [7] Carl E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.
- [8] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, March 2007.
- [9] Einar Broch Johnsen, Olaf Owe, and Marte Arnestad. Combining active and reactive behavior in concurrent objects. In *Proc. of the Norwegian Informatics Conference (NIK’03)*, pages 193–204. Tapir, November 2003.
- [10] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In David S. Wise, editor, *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI’88)*, pages 260–267. ACM Press, June 1988.
- [11] Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. Programming and deployment of active objects with application-level scheduling. In *SAC’12*, pages 1883–1888. ACM, March 2012.

- [12] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala. A comprehensive step-by-step guide*. Artima Developer, 2008.
- [13] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOP-SLA'86)*. *Sigplan Notices*, 21(11):258–268, November 1986.

# Business Process Conformance Checking Based on Event Structures

Luciano García-Bañuelos<sup>1</sup>, Nick R.T.P. van Beest<sup>2,3</sup>, Marlon Dumas<sup>1</sup> and Marcello La Rosa<sup>3,2</sup>

<sup>1</sup> University of Tartu, Estonia  
{luciano.garcia, marlon.dumas}@ut.ee  
<sup>2</sup> NICTA, Australia  
nick.vanbeest@nicta.com.au  
<sup>3</sup> Queensland University of Technology, Australia  
m.larosa@qut.edu.au

## 1 Introduction

This paper addresses the problem of business process conformance checking defined as follows: Given an event log recording the actual execution of a business process, and given a process model capturing its expected or normative execution, describe the differences between the behavior captured in the event log and that captured in the process model. In this setting, a log consists of a set of traces, where each trace is a sequence of events. An event refers to the execution of an activity in the process.

This problem has been approached using *replay* [4] and *trace alignment* [7]. Replay takes as input one trace at a time and determines the maximal prefix of the trace (if any) that can be parsed by the model. When it is found that a prefix can no longer be parsed by the model, error-recovery techniques are used to correct the parsing error and continue parsing as much as possible the remaining input trace. Trace alignment identifies, for each trace in the log, the closest corresponding trace(s) produced by the model and then highlights the points where the trace and the model diverge. However, trace alignment cannot characterize the exact differences observed in a given state of the process in a concise and understandable way, particularly for processes with a large number of possible traces.

In this abstract, we outline a method that, given a process model and an event log, returns a set of statements in natural language describing all the behavior observed in the log but not allowed by the process model (and vice versa). The method relies on a well-known model of concurrency, namely prime event structures. We show that the stated problem of conformance checking can be approached by folding the input event log into an event structure, unfolding the process model into another event structure, and comparing the two event structures via an error-correcting synchronized product.

## 2 Approach

The overall approach is depicted in Figure 1. In this section we describe each of the steps in turn.

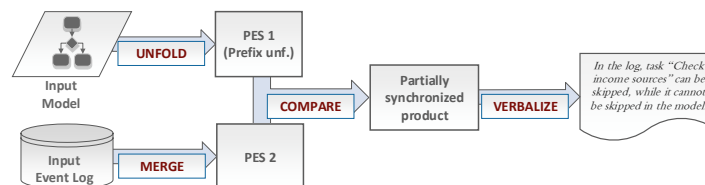


Figure 1: Overall approach

### From a log to a prime event structure

In previous work [5], we presented a method to generate a prime event structure (PES) from an event log. The method consists of two steps. First the event log, seen as a set of traces, is transformed into a set of runs by invoking a *concurrency oracle*. In essence, each trace is turned into a run by relaxing the total order induced by the trace into a partial order such that two events are not causally related if the concurrency oracle has determined that they occur concurrently. Existing concurrency oracles such as those proposed in the  $\alpha$  process mining algorithm [8] or in [1] can be used for this purpose.

Second, the runs are merged into an event structure in a lossless manner, which means that the set of maximal configurations of the event structure is equal to the set of runs. For example, consider the log in Figure 2(a), consisting of 16 traces: 3 instances  $t_1$  (cf. column “N”), 3 instances of  $t_2$ , so on. Using the concurrency oracle of the  $\alpha$  algorithm, we conclude that event classes B and C are concurrent, so that the set of runs in Figure 2(b) can be constructed. The notation  $e:A$  indicates that event  $e$  represents an occurrence of event class A in the original log. By merging events with the same label and the same history (i.e. same prefix), we obtain the PES in Figure 2(c). In this figure, the notation  $\{e_1, e_2 \dots e_i\}:A$  indicates that events  $\{e_1, e_2 \dots e_i\}$  represent occurrences of event class A in different runs.

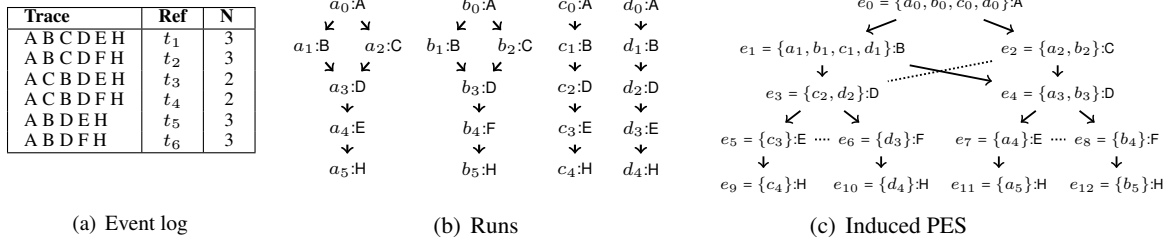


Figure 2: Example of construction of a PES from a set of traces

### From a model to a prime event structure

The proposed method takes process models as input, which are captured as Workflow nets (WF-nets) [6], i.e. Petri nets with a single start and a single end place such that every node is on a path from the start to the end. Mappings from common process modeling notations (e.g. BPMN) to WF-nets have been defined in the literature [2]. Event structures can be losslessly derived from workflow nets via unfoldings of Petri nets using well-known unfolding techniques. In the case of acyclic nets, a full unfolding can be computed and a PES can be trivially derived. In the case of bounded Petri nets with cycles, it is possible to calculate a finite prefix unfolding that captures all the behavior in the original net. A PES (prefix) can then be derived from such prefix unfolding. Several prefix unfoldings have been defined in the literature, e.g. the *complete prefix unfolding* [3].

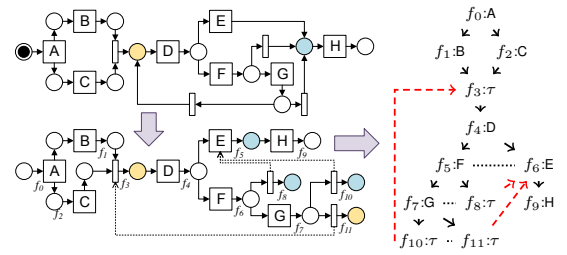


Figure 3: From a workflow net to a PES prefix

### Comparing prime event structures

The comparison of event structures is performed by means of an error-correcting synchronized product that we call a *partially synchronized product* (PSP). A PSP is built starting from empty configurations. At each step, a pair of events from each PES is matched if and only if their labels and causal order are consistent. Every unmatched event is “hidden” to let the simulation proceed. By using a heuristic search, we construct a PSP that contains the set of optimal matchings for every runs in the event log PES. Fig. 4 presents an excerpt of the PSP of the events structures from Fig. 2 and 3. The box on top corresponds to the state where configurations  $C^l = \{e_0, e_1\}$  (log PES) and  $C^r = \{f_0, f_1\}$  (model PES) have been processed, resulting in the matching  $\{(e_0, f_0)_A, (e_1, f_1)_B\}$ . Given the above state, the events  $\{e_2:C, e_3:D\}$  from log PES would be enabled, and so is  $f_2:C$  from the other PES. Thus, four possible moves are possible in the PSP: (i) the matching of events  $e_2$  and  $f_2$ , both carrying the label C, (ii) the (left) hiding of  $f_2:C$ , and (iii) the (right) hiding of  $e_2:C$  and  $e_3:D$ . Interestingly, the fragment above alone captures the fact that “*In the event log, task C can be skipped, while in the model it cannot*”. Although not illustrated, the PSP supports special-purpose moves to operate with PES prefixes.

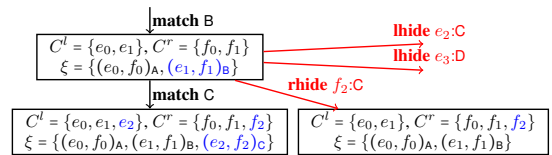


Figure 4: Fragment of PSP of  $\mathcal{E}^1$  and  $\mathcal{E}^2$

## REFERENCES

3

**Verbalizing differences**

We identify three categories of differences. The first type of difference is the one in which the labels of events confined in the mismatch context can be paired. The mismatch stems from differences in the underlying behavior relations, e.g. parallel vs. sequential relations. The second type concerns composite mismatches, in which the information of two branches in the PSP needs to be combined in order to diagnose the difference. For example, a “task skipping” is detected when an event is hidden in one branch of the PSP and there exists a sibling node where the same event is matched, and both nodes share the same parent node. The third type concerns differences comprising non-observed behavior. For instance, the model describes a cycle consisting of a set of events that cannot be found in the log, or vice versa.

Using the PSP, each change from a particular category can be verbalized to describe the exact difference between the observed log and the model. The different operations in the PSP uniquely describe the differences between the model and the log. Table 1 provides an example of the verbalization of differences with the model in Fig. 3 for each change category.

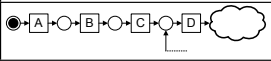
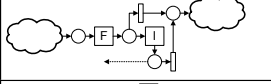
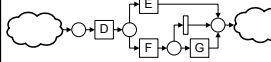
Difference	Log compared to Fig. 3	Verbalization
Type 1		In the model, B and C are in parallel, while in the log, B precedes C.
Type 2		In the log, G is substituted by I.
Type 3		In the model, the interval [D, F, G] is repeated multiple times, while in the log it is not.

Table 1: Desired verbalization of differences for each change category.

For Type 1, the PSP describes a series of mismatches concerning the same event for both PESs. When considering the context in the PESs, it becomes evident from the ordering relations that in one PES B is in parallel with C, while in the other PES B and C are causal. In the Type 2 example, on one side G is hidden, while on the other side I is hidden. From this, it can be concluded that G has been replaced with I. Finally, Type 3 concerns behavior specified in the model that is not observed in the event log. A mismatch in this category can be identified by identifying sets of events that are not present in the PSP. Of particular interest are cutoff events because they provide hints about the nature of the missing behavior. For the example shown in Table 1, the cutoff event  $f_{10}:\tau$  would not be part of the PSP, the absence of which would reveal the fact that cycle [D, F, G] is never observed in the event log.

The examples of Table 1 show that we can characterize behavior in the log that is not in the model and vice versa in a way that is understandable to users. Using this method, the conformance of an event log to a model can be assessed more compactly and precisely than existing techniques.

**References**

- [1] Jonathan E. Cook and Alexander L. Wolf. Event-based detection of concurrency. In *FSE*. ACM, 1998.
- [2] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information & Software Technology*, 50(12), 2008.
- [3] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of mcmillan’s unfolding algorithm. *Formal Methods in System Design*, 20(3), 2002.
- [4] Anne Rozinat and Wil M.P. van der Aalst. Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1), 2008.
- [5] Nick R.T.P. van Beest, Marlon Dumas, Luciano García-Bañuelos, and Marcello La Rosa. Log delta analysis: Interpretable differencing of business process event logs. In *BPM 2015*. Springer, 2015.
- [6] Wil M.P. van der Aalst. Verification of workflow nets. In *Appl. and Theory of Petri Nets 1997*. Springer, 1997.
- [7] Wil M.P. van der Aalst, Arya Adriansyah, and Boudewijn van Dongen. Replaying history on process models for conformance checking and performance analysis. *WIREs Data Min. Knowl. Discov.*, 2(2), 2012.
- [8] Wil M.P. van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: discovering process models from event logs. *IEEE TKDE*, 16(9), 2004.

# Formal Verification using Parity Games

Mathias Nygaard Justesen

Technical University of Denmark, Kongens Lyngby, Denmark  
s123152@student.dtu.dk

While many problems can be reduced to solving parity games, see [FL10] for example, verification frameworks using parity game solvers as a backend technology seem quite unexplored. In this abstract we report an initial attempt at building an infrastructure for a verification framework, which so far captures model checking for the modal  $\mu$ -calculus.

At least two toolsets, mCRL2 and LTSmin, reduce the model-checking problem to parity game solving, but they both do so by encoding the problem as a parameterized boolean equation system (PBES) [CGK<sup>+</sup>13, KvdP14]. We take a more direct, game-based approach, first proposed by Colin Stirling [Sti96, BS06], where the problem is not encoded as a PBES, but instead the system is modelled as a labelled transition system and the properties are specified as a formula in the modal  $\mu$ -calculus. This is an interesting logic, because it subsumes other widely used modal logics, such as LTL and CTL [Koz83].

## 1 Basic Notions

**Parity games.** A parity game is played by two players, called Player 0 and Player 1, on a directed graph in which all nodes are labelled with priorities. Formally, a *parity game* is a tuple  $G = (V, V_0, V_1, E, \Omega)$ , where  $(V, E)$  forms a directed, total graph. Player 0 controls the nodes in  $V_0$ , and Player 1 controls the nodes in  $V_1$ , such that  $V = V_0 \cup V_1$  and  $V_0 \cap V_1 = \emptyset$ . The priority function  $\Omega : V \rightarrow \mathbb{N}$  assigns a natural number to each node, called the *priority* of the node.

The game starts in a node  $v_0 \in V$  and an infinite sequence of nodes is constructed as follows. If the play so far has yielded a finite sequence of nodes  $v_0v_1 \dots v_i$  and  $v_i$  is in  $V_j$ , then Player  $j$  selects a node  $w$ , such that  $(v, w) \in E$ , and the play continues with the sequence  $v_0v_1 \dots v_iw$ . The winner of the play  $v_0v_1v_2 \dots$  is Player 0, if the highest priority that occurs infinitely often is even, otherwise Player 1 wins.

A *strategy* for Player  $j$  is a function  $\sigma : V^*V_j \rightarrow V$  that maps every initial play  $v_0v_1 \dots v_i$  ending a node  $v_i \in V_j$  to a successor node  $v_{i+1}$ , such that  $(v_i, v_{i+1}) \in E$ .

An important property of parity games is that they exhibit *positional determinacy* [Kü02], i.e., the set  $V$  of nodes in an arbitrary game  $G$  can be divided into two *winning regions*,  $W_0$  and  $W_1$ , such that Player  $j$  can win every game that starts in a node  $v \in W_j$  by following a *winning strategy*. Moreover, a winning strategy is also positional, i.e., there exists a function  $\sigma' : V_j \rightarrow V$  such that  $\sigma(v_0 \dots v_i) = \sigma'(v_i)$  for  $v_i \in V_j$ . Thus, an algorithm for solving parity games should compute the winning regions and the positional winning strategies.

**Labelled transition systems.** A  $(P, A)$ -labelled transition system (LTS) is a tuple  $\mathbb{S} = (S, V, R)$ , such that  $S$  is the set of states;  $V : P \rightarrow \mathcal{P}(S)$  is a *valuation*, i.e.,  $V(p)$  is the set of states where  $p$  is true; and  $R = \{R_a \subseteq S \times S \mid a \in A\}$  contains the action-labelled relations of the system.

**Modal  $\mu$ -calculus.** Given a set of proposition letters  $P$  and a set of actions  $A$ , the collection of formulas  $\varphi$  in modal  $\mu$ -calculus are defined by the following grammar

$$\varphi ::= \top \mid \perp \mid p \mid \neg p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle a \rangle \varphi \mid [a] \varphi \mid \mu x. \varphi \mid \nu x. \varphi$$



where  $p, x \in P$  and  $a \in A$ . Informally,  $\top$  and  $\perp$  are true and false, respectively;  $\neg$ ,  $\wedge$ , and  $\vee$  are the usual Boolean operators;  $\langle a \rangle$  and  $[a]$  are *modal operators*; and  $\mu$  and  $\nu$  are *fixpoint operators*.

The semantics of the modal  $\mu$ -calculus can be defined game-theoretically in terms of a so-called *evaluation game* [Ven08]. It is helpful to think of the game in terms of proving or disproving a formula  $\varphi$ . Player 0 is trying to prove  $\varphi$ , and Player 1 is trying to disprove it. The problem of determining the winner of an evaluation game can be reduced to finding the winning regions of a parity game. Furthermore, the model checker can construct examples and counter-examples from the winning strategies.

## 2 The Framework

The overall goal is to investigate the use of parity games for formal verification. Thus, an efficient solver is a critical part of a verification framework. Therefore, in the first version we use PGSolver [FL14] as the backend solver, because it allows for experimentation with different algorithms. Furthermore, a model checker for modal  $\mu$ -calculus is implemented. This implementation is based on an on-the-fly generation of a parity game on the basis of a labelled transition system and a formula in modal  $\mu$ -calculus.

Parity games of a few million nodes with average degree  $d = 3$  can be solved in less than a minute<sup>1</sup>, so speed is not an immediate concern. However, memory is quickly exhausted as the LTS or the formula becomes complicated, since the game graph consists of  $O(|S| \cdot |Sfor(\varphi)|)$  nodes.<sup>2</sup> One approach to overcome this problem is solving parity games symbolically, which leads to a more compact representation [BEKR09, KvdP14]. For a simple one-node LTS with three actions, the current implementation can check formulas with eleven alternating fixpoints in a few seconds, by generating and solving a parity game of 1.2 million nodes. For such formulas the state space grows exponentially with the depth of the formula, so a formula with twelve alternating fixpoints results in a parity game of 3.7 million nodes, which exhausts the memory of the parity game solver.

We have also considered a new type of algorithm due to Steen Vester, in which parity games are considered in a certain normal form, where the game is strictly turn-based, and where Player 0 only controls nodes of even parity and Player 1 only controls nodes of odd parity. The algorithms in consideration exploit these restrictions in order to simplify the solving process. Considering parity games in this normal form is viable, because it is possible to transform any parity game to one in normal form in linear time, such that no player has any additional strategic advantages and the winning regions are preserved. In practice, the normal-form algorithms perform worse than the state-of-the-art algorithm due to Zielonka [Zie98, FL09], but they perform well on games that they are theoretically well suited for, i.e., dense game graphs that are already in normal form (need no transformation). This is promising for the development of other specialized algorithms. These may not perform better in general, but in special cases of interest, e.g., an algorithm that exploits the tree-like structure of parity games constructed from evaluation games.

Overall, this is promising for using parity games for other verification techniques, e.g., controller synthesis, which amounts to finding the winning strategies in a parity game [RW89].

<sup>1</sup>Benchmarks carried out on a machine with four 3.5 GHz Intel Core i5 processors and 8 GB RAM space. The implementation does not support parallel computations, hence, the benchmarks was only run on one processor.

<sup>2</sup> $Sfor(\varphi)$  denotes the subformulas of  $\varphi$ .

## References

- [BEKR09] Marco Bakera, Stefan Edelkamp, Peter Kissmann, and Clemens D. Renner. Solving  $\mu$ -calculus parity games by symbolic planning. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *Lect. Notes Comput. Sci.*, 5348:15–33, 2009.
- [BS06] Julian Bradfield and Colin Stirling. Modal  $\mu$ -calculi. *Handbook of modal logic*, 3:721 – 756, 2006.
- [CGK<sup>+</sup>13] Sjoerd Cranen, Jan Friso Groote, Jeroen JA Keiren, Frank PM Stappers, Erik P de Vink, Wieger Wesselink, and Tim AC Willemse. An overview of the mCRL2 toolset and its recent advances. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213. Springer, 2013.
- [FL09] Oliver Friedmann and Martin Lange. Solving parity games in practice. In Zhiming Liu and Anders P. Ravn, editors, *Automated Technology for Verification and Analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 182–196. Springer Berlin Heidelberg, 2009.
- [FL10] Oliver Friedmann and Martin Lange. Local strategy improvement for parity game solving. *arXiv preprint arXiv:1006.1409*, 2010.
- [FL14] Oliver Friedmann and Martin Lange. The PGSOLVER collection of parity game solvers — version 3. URL: <https://github.com/tcsprojects/pgsolver/raw/master/doc/pgsolver.pdf>, 2014.
- [Koz83] Dexter Kozen. Results on the propositional  $\mu$ -calculus. *Theoretical Computer Science*, 27(3):333 – 354, 1983. Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982.
- [KvdP14] Gijs Kant and Jaco van de Pol. Generating and solving symbolic parity games. In *Proceedings 3rd Workshop on GRAPH Inspection and Traversal Engineering, GRAPHITE 2014, Grenoble, France, 5th April 2014.*, pages 2–14, 2014.
- [Kü02] Ralf Küsters. Memoryless determinacy of parity games. In Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors, *Automata Logics, and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*, pages 95–106. Springer Berlin Heidelberg, 2002.
- [RW89] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, Jan 1989.
- [Sti96] Colin Stirling. *Modal and temporal logics for processes*. Springer, 1996.
- [Ven08] Yde Venema. Lectures on the modal  $\mu$ -calculus. *Institute for Logic, Language and Computation, University of Amsterdam*, 2008.
- [Zie98] Wiesław Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135 – 183, 1998.

# Join inverse categories and reversible recursion

Robin Kaarsgaard

DIKU, Department of Computer Science, University of Copenhagen  
Copenhagen, Denmark  
robin@di.ku.dk

Many reversible functional programming languages (such as Theseus [8] and the combinator calculi  $\Pi$  and  $\Pi^0$  [3]) as well as categorical models thereof (such as  $\dagger$ -traced symmetric monoidal categories [3]) come equipped with a tacit assumption of totality, a property that is neither required [2] nor necessarily desirable as far as guaranteeing reversibility is concerned. Shedding ourselves of this assumption, however, requires us to handle partiality explicitly as additional categorical structure.

One approach which does precisely that is Cockett & Lack's notion of *inverse categories* [4], a specialization of *restriction categories*, which have recently been suggested and developed by Giles [5] as models of reversible (functional) programming. In this paper, we will argue that assuming ever slightly more structure on these inverse categories, namely the presence of *countable joins* of parallel morphisms, gives rise to a number of additional properties useful for modelling reversible functional programming, notably reversible (tail) recursion and recursive data types (via  $\omega$ -algebraic compactness with respect to structure-preserving functors), which are not otherwise present in general. This is done by adopting two different, but complementary, views on inverse categories with countable joins as enriched categories – as CPO-categories, and as (specifically  $\Sigma$ Mon-enriched) unique decomposition categories.

**Background** In the framework of restriction categories, partiality is handled by equipping each morphism  $f : A \rightarrow B$  with a partial identity morphism  $\bar{f} : A \rightarrow A$  (the *restriction idempotent* of  $f$ , intuitively the identity defined precisely where  $f$  is defined) subject to a few axioms, notably that  $\bar{f}$  is the right-identity of  $f$  under composition. This definition provides a partial ordering on Hom sets by defining  $f \leq g$  for parallel morphisms  $f$  and  $g$  iff  $g \circ \bar{f} = f$ .

Joins on morphisms (see, e.g., Guo [6]) are then defined to be joins with respect to this partial order (subject to a few axioms), with the caveat that parallel morphisms  $f$  and  $g$  can only be joined if they are *join compatible*, which they are iff  $g \circ \bar{f} = f \circ \bar{g}$  (intuitively, if they agree on all points in their domain where they are both defined). This definition is then straightforwardly extended to sets (in this particular case, countable ones) of parallel morphisms by saying that a set  $S \subseteq \text{Hom}(A, B)$  is join compatible if all morphisms of  $S$  are pairwise join compatible. A restriction category is thus said to have (countable) joins if all (countable) join compatible sets have a join, and the category has a *restriction zero object*, that is, a zero object in the usual sense which additionally satisfies that the zero map  $0_{A,A} : A \rightarrow A$  is a restriction idempotent (i.e., that  $0_{A,A} = \overline{0_{A,A}}$ ) for all objects  $A$  (the zero map  $0_{A,B}$  is the unit for joins in  $\text{Hom}(A, B)$ ).

Perhaps more immediately important to our applications, restriction categories allow for a definition of a *partial isomorphism* as a morphism  $f : A \rightarrow B$  for which there exists a *partial inverse*  $f^* : B \rightarrow A$  such that  $f^* \circ f = \bar{f}$  and  $f \circ f^* = \overline{f^*}$ . An *inverse category* is then defined to be a restriction category in which all morphisms are partial isomorphisms; as such, inverse categories are “groupoids with partiality,” and can be canonically equipped with the structure of a  $\dagger$ -category by letting the  $\dagger$ -functor map each morphism to its partial inverse. Keeping with this canonical structure, we will use  $f^\dagger$  for the partial inverse of  $f$  from here on out. Inverse categories can be equipped with joins in the same way as general restriction categories can (with slightly more work, see Guo [6]).

**CPO-enrichment** Since inverse categories come equipped with partially ordered Hom sets, demonstrating CPO-enrichment reduces to producing suprema of  $\omega$ -chains and showing that composition is continuous and strict. Let  $\mathcal{C}$  be an inverse category with countable joins. For an  $\omega$ -chain  $\{f_i\}_{i \in \omega}$  of some  $\text{Hom}_{\mathcal{C}}(A, B)$ , we define its supremum by<sup>1</sup>

$$\sup \{f_i\}_{i \in \omega} = \bigvee_{i \in \omega} f_i .$$

That this join exists follows from the fact that  $f \leq g$  implies that  $f$  and  $g$  are join compatible. That composition is continuous follows directly by this definition since

$$g \circ \bigvee_{f \in F} f = \bigvee_{f \in F} (g \circ f) \quad \text{and} \quad \left( \bigvee_{f \in F} f \right) \circ h = \bigvee_{f \in F} (f \circ h)$$

are axioms of joins [6]; similarly, strictness of composition follows by the universal mapping property for the zero object, noting that the zero map  $0_{A,B}$  is least in the partial order on  $\text{Hom}_{\mathcal{C}}(A, B)$  for all objects  $A, B$ .

From this follows the existence of fixed points for all continuous *morphism schemes for recursion*, i.e., monotone and continuous functions of the form  $f : \text{Hom}_{\mathcal{C}}(A, B) \rightarrow \text{Hom}_{\mathcal{C}}(A, B)$  by Kleene's fixed point theorem, and can thus be used to model recursion. A further pleasant property is *local continuity* of the canonical  $\dagger$ -functor on  $\mathcal{C}$ , i.e., the map  $\text{inv}_{A,B} : \text{Hom}_{\mathcal{C}}(A, B) \rightarrow \text{Hom}_{\mathcal{C}}(B, A)$  given by  $\text{inv}_{A,B}(f) = f^\dagger$  is monotone and continuous for all objects  $A, B$ .

Combining the two, we can show that each continuous morphism scheme for recursion  $f$  has a *fixed-point adjoint*  $f_{\ddagger}$  such that  $(\text{fix } f)^\dagger = \text{fix } f_{\ddagger}$ ; intuitively, that the partial inverse of a recursive function can be constructed recursively in a canonical way. This is done by defining

$$f_{\ddagger} = \text{inv}_{A,B} \circ f \circ \text{inv}_{B,A}$$

which is continuous since it is a continuous composition of continuous functions;  $\text{fix } f_{\ddagger} = (\text{fix } f)^\dagger$  can then be shown using local continuity of the  $\dagger$ -functor, and by noting that  $f_{\ddagger}^n = \text{inv}_{A,B} \circ f^n \circ \text{inv}_{B,A}$ . This gives us reversible recursion in the style of RFUN [9]: a recursive function is inverted by replacing recursive calls with calls to the inverse function, and then inverting the remainder of the function. Further, by considering more general morphism schemes, we can get a procedure for representing *parameterized functions* in the style of Theseus [8].

Another consequence is the fact that every inverse category with countable joins can be embedded (fully faithfully and in a join and restriction preserving manner) into an inverse category that is  $\omega$ -algebraically compact with respect to the class of join and restriction preserving functors. The proof of this theorem is somewhat involved: it relies on a coincidence between *restriction monics* in inverse categories (split monics that split a restriction idempotent) and *embeddings* in CPO-categories (morphisms  $f : A \rightarrow B$  with *projections*  $f^* : B \rightarrow A$  such that  $f^* \circ f = 1_A$  and  $f \circ f^* \leq 1_B$ ); on Guo's characterization of join restriction categories as partial map categories with certain stable colimits [6]; and on Adámek's fixed point theorem [1].

**$\Sigma$ Mon-enrichment and unique decomposition** Another way to approach inverse categories with countable joins is as  $\Sigma$ -monoid enriched categories in the sense of Haghverdi [7]. Briefly, a  $\Sigma$ -monoid consists of a set  $S$  equipped with a partial sum function  $\Sigma$  defined on

<sup>1</sup>This is a slight abuse of notation since joins in restriction categories are unordered, i.e., defined on sets rather than families. The join  $\bigvee_{i \in \omega} f_i$  should thus be taken to mean  $\bigvee_{f \in F} f$  where  $F = \{f_i \mid i \in \omega\}$ .

countable families of  $S$  (say that such a family is *summable* if its sum exists), subject to the axioms of *partition-associativity* (a family is summable iff any partitioning of it is piecewise summable, and the sum of the pieces coincide with the sum of the family) and *unary sum* (the sum of a singleton family is equal to its element). It is straightforwardly the countable joins in inverse categories satisfies these axioms, with summability coinciding with join compatibility.

If we, in addition, suppose that our inverse category  $\mathcal{C}$  with countable joins is equipped with a *disjoint sum tensor* in the sense of Giles [5] (a symmetric monoidal restriction functor  $\cdot \oplus \cdot$  with the restriction zero as unit, and equipped with jointly epic injections  $\Pi_1 : A \rightarrow A \oplus B$  and  $\Pi_2 : B \rightarrow A \oplus B$  and jointly monic coinjections  $\Pi_1^\dagger : A \oplus B \rightarrow A$  and  $\Pi_2^\dagger : A \oplus B \rightarrow B$ ), we get straightforwardly that  $\mathcal{C}$  is a *unique decomposition category* (a symmetric monoidal category with *quasi-injections*  $\iota_j : X_j \rightarrow \bigoplus_I X_i$  and *quasi-projections*  $\rho_j : \bigoplus_I X_i \rightarrow X_j$  for all  $j \in I$  where  $I$  a finite index set, subject to a two axioms [7]). Using join compatibility of disjoint morphisms, it follows by Haghverdi [7] that  $\mathcal{C}$  is traced, and that the trace can be constructed by

$$\text{Tr}_{A,B}^U(f) = f_{11} + \sum_{n \in \mathbb{N}} f_{21} \circ f_{22}^n \circ f_{12} = f_{11} \vee \bigvee_{n \in \mathbb{N}} f_{21} \circ f_{22}^n \circ f_{12}$$

for all  $f : A \oplus U \rightarrow B \oplus U$ , where  $f_{ij} = \rho_j \circ f \circ \iota_i = \Pi_j^\dagger \circ f \circ \Pi_i$ . In this special case, however, this is not just a trace, but a  $\dagger$ -trace (*i.e.*, it satisfies  $\text{Tr}_{B,A}^U(f^\dagger) = \text{Tr}_{A,B}^U(f)^\dagger$ ): this can be seen by realizing that  $(f_{ij})^\dagger = f_{ji}^\dagger$ , and by using  $(\bigvee_{f \in F} f)^\dagger = \bigvee_{f \in F} f^\dagger$  which follows directly from local continuity of the  $\dagger$ -functor in the CPO-view. This is significant given that  $\dagger$ -traces are used to model reversible tail recursion.

**Conclusion** The existence of countable joins in inverse categories provides us with a model of partial reversible functional programming with recursive types and general recursion in the style of RFUN. Further assuming the existence of a disjoint sum tensor allows us to extend the standard model of  $\dagger$ -traced symmetric monoidal categories to one with a notion of partiality.

## References

- [1] Jiří Adámek. Recursive Data Types in Algebraically  $\omega$ -Complete Categories. *Information and Computation*, 118:181–190, 1995.
- [2] Holger Bock Axelsen and Robert Glück. What do reversible programs compute? In *Foundations of Software Science and Computational Structures*, volume 6604, pages 42–56, 2011.
- [3] William J. Bowman, Roshan P. James, and Amr Sabry. Dagger traced symmetric monoidal categories and reversible programming. In *Reversible Computation*, volume 7165 of *LNCS*, pages 51–56, 2011.
- [4] J. R. B. Cockett and Stephen Lack. Restriction categories I: Categories of partial maps. *Theoretical Computer Science*, 270(2002):223–259, 2002.
- [5] Brett Gordon Giles. *An investigation of some theoretical aspects of reversible computing*. PhD thesis, University of Calgary, 2014.
- [6] Xiuzhan Guo. *Products, Joins, Meets, and Ranges in Restriction Categories*. PhD thesis, University of Calgary, 2012.
- [7] Esfandiar Haghverdi. Unique decomposition categories, Geometry of Interaction and combinatory logic. *Mathematical Structures in Computer Science*, 10(2):205–230, 2000.
- [8] Roshan P. James and Amr Sabry. Theseus: A High Level Language for Reversible Computing. Available at <http://www.cs.indiana.edu/~sabry/papers/theseus.pdf>, 2014.
- [9] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a reversible functional language. In *Reversible Computation*, volume 7165 of *LNCS*, pages 14–29, 2012.

# Errors as Data Values as the Language Default <sup>\*</sup>

Tero Hasu and Magne Haveraaen

Bergen Language Design Laboratory  
 Department of Informatics  
 University of Bergen, Norway  
<http://www.i.i.uib.no/~{tero,magne}>

## Abstract

A “thrown” exception is a non-local side effect that complicates static reasoning about code. In some programs errors are instead propagated as ordinary values. Such propagation is sometimes done in monadic style, and some languages include syntactic conveniences for writing expressions in that style. We sketch a language-based failure management approach in which error-monad-resembling transparent error value propagation is made the language *default*. The approach accommodates language designs with all-referentially-transparent expressions, and syntactic conveniences resembling those of traditional exception mechanisms. Our proof-of-concept implementation of the approach is furthermore capable of automatically checking data invariants and function pre- and post-conditions, recording a trace of the failed or unevaluatable expressions caused by an error, and in some cases retaining “bad” values for potential use in recovering from an error.

## 1 Bad-Value-Extended Data Types as a Convention

A number of mainstream languages include a `try/catch`-style facility for intercepting non-local-returning, exceptional control transfers triggered by errors. Propagating error information using such mechanisms comes at the cost of making static reasoning about code harder. As exceptions are side effects, *referential transparency* (RT) of expressions that may throw is lost; this means that replacing an expression by its value may not preserve program semantics, as the value may not capture everything that the evaluation of an expression does.

A more traditional alternative is to report errors through return values, by having some values of the return type signify an error. While this approach preserves RT, it also tends to involve tedious, explicit checking and propagation of error return values. One may be able to encapsulate the tedious work within an abstraction, but the abstraction can only encompass operations for which there is a known way of determining which return values indicate an error. A simple way to enable such determination is to augment (in a general way) each return type to have a carrier set that is a disjoint union of values that are explicitly good or bad.

For example, for a potentially-failing Haskell function whose successfully computed values are of type `Integer`, we might specify the return type `Result Integer`, with the type constructor `Result` defined as shown below. In the definition, the parameter `t` is the type of the good “payload,” and `Error` is the type of an error information object.

```
data Result t = Good t | Bad Error
```

`Result` can be made into an error monad by defining the operations of the `Monad` type class with the appropriate semantics. The monad then encapsulates implicit actions to check for `Good` function arguments, and to propagate any `Bad` arguments while skipping subsequent,

---

<sup>\*</sup>This research has been supported by the Research Council of Norway through the project DMPL—Design of a Mouldable Programming Language.

unevaluatable function applications. It is not especially convenient to write monadic expressions such as `mv >>= (\x -> f x >>= (\y -> g y))`, but with Haskell’s `do` notation as syntactic sugar one may instead write `do x <- mv; y <- f x; g y`. This is more convenient, but not uniform; the programmer has both monadic and “bare” values to deal with, and different syntaxes for monadic and non-monadic expressions (cf. the above expressions versus `g (f mv)`).

We note that the `Result` type constructor is universal: any type can be augmented with a set of error values by “wrapping” it in `Result`. What if we made use of this power for purposes of uniformity, and adopted a language-wide convention of computing with `Result`-wrapped values? We have been exploring this question, by devising and experimenting with various language-integrated mechanisms to support such a convention.

Two particularly useful, orthogonal kinds of such language-based support are transparent propagation of error values, and automated adaptation to other error reporting conventions, which we discuss in sections 1.1 and 1.2, respectively. Designing and implementing such support is made easier by consistency of convention and explicit badness of values: it is sufficient to only have “monadic” expression syntax, which can for the most part be made to look conventional, while still having it express `Result` processing; and generating code to check for a bad value requires no context, but merely the insertion of a single, known predicate check.

Our approach seems most promising for languages focused on supporting static reasoning, in that such languages benefit from RT preservation and simple, uniform rules for error management. Furthermore, static reasoning opportunities facilitate optimizations to reduce the readability-hampering code bloat that results from naive implementations of the approach.

We have implemented our language-integrated error management scheme in the form of `Erda`<sup>1</sup>, which is a family of experimental programming languages that have a Racket-resembling syntax. `ErdaRVM` is a dynamically typed language targeting the Racket virtual machine (VM), while `ErdaC++` is a statically typed language that compiles to C++ source code. The implementation of `ErdaRVM` is purely based on program transformations expressed in terms of the host language’s macros, whereas `ErdaC++` additionally relies on a compiler.

## 1.1 Uniform, Language-Wide Error Propagation

Assuming that every operation (whether user-defined or built-in) consistently reports any failures via distinctly bad return values, then what remains is to add language support for transparent processing of values that make such an explicit distinction. In our solution the processing includes checking for bad values, skipping operations that may not be performed, and recording information about failures and skipped operations inside the processed values, for the benefit of error reporting and recovery. The processing is performed by (highly portable) code that the language implementation emits between operations that appear in a program; that code is similar to the implicit actions encoded in terms of the primitives of an error monad.

In `ErdaRVM`, the literal syntax `0` expresses the constant value (`Good 0`). The `factorial` function, as below, transparently processes `Result` values throughout; as shown here, it correctly obeys the convention of reporting errors (in this case its inability to compute  $x!$  for any  $x < 0$ ) by returning a `Bad` value, which can be instantiated with `raise`:

```
(define (factorial x)
  (cond
    [(< x 0) (raise 'bad-arg)]
    [(= x 0) 1]
    [else (* x (factorial (- x 1)))]))
```

<sup>1</sup>Code and documentation for `Erda` is available at <https://www.ii.uib.no/~tero/erda-2015/>

In addition to `raise`, `ErdaRVM` includes a number of other constructs (e.g., a `try/catch`-inspired `try`, and a less familiar `on-alert` [1]) capable of dealing with `Bad` values. `Bad` values may be stored in variables as normal. By default, a `defined` function only receives `Good` arguments<sup>2</sup>; for example, the expression `(factorial (factorial -1))` only causes `factorial` to be called once, as `ErdaRVM` skips the outer call due to its `Bad` argument.

In effect, `ErdaRVM` has extended the implementation of `factorial` to allow the normal control flow of the program to proceed irrespective of whether the result is a good value or a bad value. If it is a good value, the computation will proceed as normal. If the result is a bad value, the computation can accumulate information about how the value should have been processed after its inception. In contrast, an error monad has no access to such contextual information; a language implementation can access even the uncomputable expression itself.<sup>3</sup>

## 1.2 Declarative Adaptation to Other Conventions

Our language-native error reporting convention concerns both natively defined functions and foreign primitives. Due to our wholesale adoption of the convention, we wanted to support a declarative way of statically generating adaptation code for interfacing with functions that follow other conventions. Our declaration-based abstraction over foreign conventions is modeled after *alerts*, as detailed by Bagge et al [1]. An alert may be triggered due to a broken pre- or post-condition or a thrown exception, as declared. Alerts are propagated as bad data values.

Racket functions can be called from `ErdaRVM` directly, with automatic `Result` (un)wrapping of arguments and return values, but any associated alerts have to be `declared`:

```
(declare (/ x y) #:alert ([div-by-0 on-throw exn:fail:contract:divide-by-zero?]))
```

The alert facility is also useful within `ErdaRVM`. For example, rather than invoking `raise` explicitly within `factorial`, we can make it conformant by declaring an alert for it:

```
(define (factorial x) #:alert ([bad-arg pre-when (< x 0)]) ——)
```

Guarded algebras [2] serve as the formal basis for our declarative checking of failure conditions. Formally, all the applicable data invariants (which may be defined for `ErdaRVM` data types) and alert declarations together induce an idealized guard predicate for every expression. In practice we need not infer a complete pre-condition for an operation, but can rather have a code generator precede or surround or follow an operation invocation with all the appropriate individual guard expressions, in order to detect or catch errors. At the same time we can also insert code to ensure that the appropriate wrapper data type is used for the result, with available information about any error embedded into the wrapper.

## References

- [1] Anya Helene Bagge, Valentin David, Magne Haveraaen, and Karl Trygve Kalleberg. Stayin' alert: Moulding failure and exceptions to your needs. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*, pages 265–274, Portland, Oregon, October 2006. ACM Press.
- [2] Magne Haveraaen and Eric G. Wagner. Guarded algebras: Disguising partiality so you won't know whether it's there. In *Recent Trends In Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 3–11. Springer-Verlag, 2000.

<sup>2</sup>`ErdaRVM` also supports declaration of `#:handler` functions accepting `Bad` arguments.

<sup>3</sup>For a formal basis, one might consider `factorial` as defining a partial algebra for good arguments, with `ErdaRVM` doing a free term completion of the algebra for the bad arguments.



# Discounted Duration Calculus

Martin Fränzle<sup>1\*</sup>, Michael R. Hansen<sup>2†</sup> and Heinrich Ody<sup>1‡</sup>

<sup>1</sup> Department of Computing Science, University of Oldenburg, Germany  
 fraenzle@informatik.uni-oldenburg.de, heinrich.ody@uni-oldenburg.de

<sup>2</sup> DTU Compute, Technical University of Denmark mire@dtu.dk

## 1 Introduction

In economics discounting represents that money earned soon can be reinvested earlier and hence yields more revenue than money earned later. Discounting has been introduced into temporal logics to represent that something happening earlier is more important than similar events happening later [DFH<sup>+</sup>04, ABK14]. A typical example is a rail road crossing. Consider the property ‘eventually the gates are open’. While a controller leaving the gates closed an hour after the train has passed might be safe and alive, it is not useful. We can use discounting to express that the controller should not wait unnecessarily long before opening the gates. In [ABK14] this has been described as quantifying the temporal quality of a system.

So far discounting in logics only has been studied for discrete-time temporal logics (LTL, CTL\*,  $\mu$ -calculus) [DFH<sup>+</sup>04, ABK14]. Here, we study discounting in the dense temporal case. We extend Duration Calculus (DC) [CHR91] with discounting and give some examples on which we want to apply our new logic.

## 2 Discounted Duration Calculus

We use an adapted version of Duration Calculus (DC), where the chop operator is replaced by left and right neighbourhood modalities. As atomic formulae, we only allow comparison of durations with constants.

**Definition 1.** *Let  $d \in [0, 1] \cap \mathbb{Q}$ ,  $c \in \mathbb{Q}$  and let  $P$  be a proposition. Then the syntax of our fragment of DC is defined as*

$$\begin{aligned} \phi &::= \diamond_1^d \phi \mid \diamond_r^d \phi \mid \int^d S \geq c \mid \int^d S > c \mid \neg \phi \mid \phi \vee \phi \mid [S] \text{ ,} \\ S &::= P \mid \neg S \mid S \wedge S \text{ .} \end{aligned}$$

The semantics of DC is defined in terms of trajectories. A trajectory is a function

$$tr : \mathbb{R}_{\geq 0} \rightarrow \text{Varname} \rightarrow \mathbb{B}$$

assigning to each time instant and each variable a truth value.

The semantics does not just determine whether the model satisfies the formula, but rather to what extent the model satisfies the formula. Hence, we associate to a formula and a trajectory

\*Work of the author was partially supported by Deutsche Forschungsgemeinschaft within the Transregional Collaborative Research Center SFB/TR 14 AVACS.

†Work of the author was partially supported by the Danish Research Foundation for Basic Research within the IDEA4CPS project.

‡Work of the author is supported by the Deutsche Forschungsgemeinschaft (DFG) within the Research Training Group DFG GRK 1765 SCARE.

a *satisfaction value* in the form of a real number in the interval  $[0, 1]$ . The right neighbourhood modality  $\diamond_r^d$  expresses that right of the current interval there is an adjacent interval satisfying the subformula. The discount  $d$  is used to reduce the satisfaction value w.r.t. the length of the adjacent interval. The left neighbourhood modality is defined similarly. Discounting in the integral formula represents that it takes at least  $c$  units of time to satisfy the comparison. Hence, the first  $c$  time units should not be discounted.

**Definition 2.** Given a formula, a trajectory  $tr$ , and a time interval  $[k, m]$ , the semantics is defined as

$$\begin{aligned} \mathcal{I}[\diamond_r^d \phi](tr, [k, m]) &= \sup_{l \geq m} \{d^{l-m} \cdot \mathcal{I}[\phi](tr, [m, l])\} \\ \mathcal{I}[\diamond_l^d \phi](tr, [k, m]) &= \sup_{l \leq k} \{d^{k-l} \cdot \mathcal{I}[\phi](tr, [l, k])\} \\ \mathcal{I}[\int^d S \geq c](tr, [k, m]) &= \begin{cases} 0 & \text{if } \int_{t=k}^m S(t) dt < c \\ d^{m-k-c} & \text{otherwise} \end{cases} \\ \mathcal{I}[\int^d S > c](tr, [k, m]) &= \begin{cases} 0 & \text{if } \int_{t=k}^m S(t) dt \leq c \\ d^{m-k-c} & \text{otherwise} \end{cases} \\ \mathcal{I}[\neg \phi](tr, [k, m]) &= 1 - \mathcal{I}[\phi](tr, [k, m]) \\ \mathcal{I}[\phi_0 \vee \phi_1](tr, [k, m]) &= \max\{\mathcal{I}[\phi_0](tr, [k, m]), \mathcal{I}[\phi_1](tr, [k, m])\} \\ \mathcal{I}[\ulcorner \phi \urcorner](tr, [k, m]) &= \begin{cases} 1 & \text{if } \int_{t=k}^m S(t) dt = m - k \text{ and } m > k \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

If we want to use the modalities without discounting we use a discount of 1. In this case we do not explicitly write the discount. Additionally, we define as abbreviations modalities for every right and left adjacent interval and another modality for every interval:

$$\square_r \phi \stackrel{\text{def}}{=} \neg \diamond_r \neg \phi \qquad \square_l \phi \stackrel{\text{def}}{=} \neg \diamond_l \neg \phi \qquad \square \phi \stackrel{\text{def}}{=} \square_r \square_l \square_l \square_r \phi$$

## 3 Examples

### 3.1 Call Centre

Consider a customer calling a call centre with a request (see Figure 1). Let  $S$  be a state variable indicating whether the customer is waiting or interacting with the employee, let  $c$  be the time of interaction between the customer and the employee required to complete the request and let  $d$  be the factor of discounting or inflation representing the impatience of the customer. Then a high satisfaction value of the formula

$$\diamond_r \int^d S \geq c ,$$

by the model of the call centre indicates an efficient handling of requests.

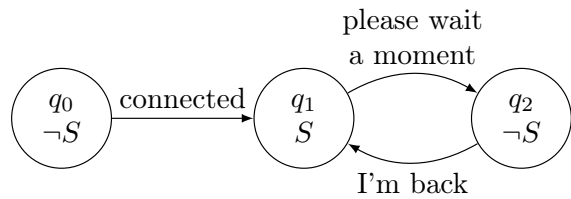


Figure 1: Automaton modelling a call to a call centre. In state  $q_0$  the customer waits to be connected to an employee. In state  $q_1$  the customer interacts with the employee. In state  $q_2$  the employee interacts with his colleagues

### 3.2 Railway Crossing

Usually at a railway crossing the gates would close when a train is approaching. However, when the street is heavily used the throughput of the street might be too low when the gates are closed whenever a train is approaching. Instead, there could be interlocks for the railway that are nlocked whenever the gates for the street are open. When a train is approaching the gates of the street should however be closed soon. This is expressed as

$$\Box([\text{apr} = 1] \implies \Diamond_r^{0.9}[\text{SG.closed}]) ,$$

where  $\text{apr}$  is the number of trains approaching and  $\text{SG}$  stands for street gate. If another train approaches, e.g. on a parallel track or behind the first train, the urgency is increased, which can be expressed by reducing the discount as in

$$\Box([\text{apr} = 2] \implies \Diamond_r^{0.8}[\text{SG.closed}]) .$$

### 3.3 Energy Consumption

Consider a system with an energy capacity  $c$ , dissipating energy while operational, but conserving energy in idle mode. Examples are distributed sensors or mobile robots. Let the state variable  $S$  describe whether the system currently is operating. Then the satisfaction value of

$$\Box_r^{0.9}(fS \leq c)$$

becomes the higher, the longer the system has not used up its energy budget.

## 4 Outlook

We introduced discounted DC and showed some small examples for which interesting properties can be expressed in our logic. Next we are interested in meaningful decision problems that can be answered automatically. For this we consider threshold satisfaction as in [ABK14], i.e. for a formula  $\phi$  and a trajectory  $tr$ , an initial interval  $[k, m]$  and  $v \in ]0, 1[$  we ask

$$\mathcal{I}[\phi](tr, [k, m]) \geq v .$$

For formulas of the form such as in the call centre example the inequality becomes  $d^\delta \geq v$ , where  $\delta$  is the length of the interval that satisfies the formula. Then  $\delta = \log_d v$  is the maximal  $\delta$  such that the inequality is still satisfied. Hence, we only have to check a bounded part of the possibly infinite trace. If we additionally assume that the number of state changes in any finite interval is bounded by some constant then threshold satisfaction seems decidable, because there is a bound on the maximal number of state changes we need to consider. For more general formulae, we will investigate approximability of maximal satisfaction values, as facilitated by discounting the far future and past.

## References

- [ABK14] Shaull Almagor, Udi Boker, and Orna Kupferman. Discounting in LTL. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 424–439. Springer, 2014.
- [CHR91] Zhou Chaochen, Charles A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information processing letters*, 40(5):269–276, 1991.
- [DFH<sup>+</sup>04] Luca De Alfaro, Marco Faella, Thomas A. Henzinger, Rupak Majumdar, and Mariëlle Stoelinga. *Model checking discounted temporal properties*. Springer, 2004.

# Algebraic Combinators for Data Dependencies and Their Applications

Eva Burrows\*

Bergen Language Design Laboratory  
Department of Informatics  
University of Bergen, Norway  
<http://bldl.iib.uib.no>

## Abstract

The notion of Data Dependency Algebra (DDA) is an algebraic formalism that turns data dependencies into first class citizens in the program code through a dedicated Application Programming Interface (API). This forms the basis of a platform independent parallel programming model [BH09]. In this paper, we further expand the theory of DDAs by proposing algebraic combinators operating on top of DDAs as a means to declare compound DDAs of custom complexity. The purpose is to allow the programmer to combine existing DDA implementations via high-level language constructs using simple declarations. The implementation of the compound DDA is generated at compile-time yet through the API its components are readily available for the programmer after declaration. We instantiate these ideas through the case-study of a DDA-based polynomial multiplication.

## Introduction

Dependence analysis is a complex process through which a compiler collects relevant information about the execution-order of program statements [Ban96]. The aim is to identify situations when statements can be reordered for optimisation purposes without changing the semantics of the program. This may lead to improved instruction scheduling with decreased number of stalls, better exploitation of instruction-level parallelism when the hardware supports it, or improved memory locality, etc. With the appearance of early parallel computing systems, compilers also met the challenge of how to generate parallel executable on demand. This formed the basis of the concept known as automatic parallelization. While most modern compilers have successfully adopted optimisation techniques based on dependence analysis, its applicability for automatic parallelization has remained limited. Dependence analysis is NP-complete in the presence of recursion, indirect addressing, pointers, or when the behaviour of the program is dynamically determined, for instance, loops with non-fixed iteration spaces, or algorithms with input-dependent dynamic dependencies. In addition, a major problem with automatic parallelization is that sequential and parallel versions of an algorithm are fundamentally different. They are based on solution paradigms that do not necessarily relate to each other. Compiler transformations, on the other hand, generally preserve the solution paradigm. Hence, dependence analysis of sequential code cannot supply sufficient knowledge to aid the complex task of parallelization: parallel task decomposition, load balancing, data distribution, synchronisation, etc.

Over the last decades, research in parallel programming models and compilers has shown that a different coding technique is required when the aim is to execute certain parts of a computation in parallel. Be that through the means of language constructs with a dedicated parallel (or concurrent)

---

\*This research has been supported by The Research Council of Norway through the project Design of a Mouldable Programming Language.

execution model, using for instance threads, parallel loops, data-parallel constructs, skeletons, message passing, or based on directives which instruct the compiler that annotated parts of the code can be executed in parallel, or based on other abstractions that help the compiler in the parallelization process.

With all that, parallel programming has proved to be very difficult and error-prone. Portability across multiple platforms and flexibility is also a major issue. Today, this is even more accentuated by the fact that applications need to be parallelized to adapt to the rapidly growing and versatile realm of parallel hardware systems. This applies to all range of computing systems, from commodity computers via embedded systems up to supercomputers. Multi-cores, many-cores, and accelerators like Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) are becoming standard yet the search for parallel programming models that meet the requirements of portability, flexibility, efficiency, scalability and programming productivity across these platforms is still ongoing.

## Parallel Programming with Data Dependencies

We pointed out that automatic dependence analysis cannot provide fine-grained details about the data dependencies occurring in a computation. Nonetheless, it is the flow of data and the presence or lack of dependencies between computational steps which determine any parallel execution. Therefore, we set our focus on fine-grained data dependencies.

The notion of *Data Dependency Algebra* (DDA), introduced in [Hav00], is an algebraic formalism that allows the programmer to present the data dependency graph of a computation as program code to a compiler. The abstraction is powerful enough to serve as the basis of a platform independent parallel programming model ensuring flexibility, productivity and portability across the platforms [BH09]. The approach also provides a high and easy to manipulate level for the programmer to deal with data distribution and placement [BH12]. In general, DDA-based parallel code generation is doable for any parallel systems with a well defined space-time communication structure. This has been shown for shared- and distributed-memory model computers, GPUs and FPGAs [Sør98, BH09, Bur14].

Central to this approach is the ability to extract manually the data dependency graph of an algorithm and code the computation in terms of the DDA API. This approach primarily suits computations with static and scalable data dependencies where the patterns are regular. Some data dependencies are more complex and probably less regular than for instance the butterfly pattern of the Fast Fourier Transform, sorting networks, or stencil computations of PDE solvers. Coding complex dependencies may easily become cumbersome.

In this presentation, we propose algebraic combinators operating on top of DDAs as a means to declare compound DDAs of custom complexity. The purpose is to allow the programmer to combine existing, easy to code DDA implementations via high-level language constructs. The implementation of the compound DDA is generated at compile-time yet through the API its components are readily available for the programmer after declaration. The following combinators are presented:

1. The *parallel DDA combinator* allows DDAs to be placed next to each other resulting in a larger DDA, to be referenced as a standalone DDA, without defining any additional connection between them.
2. On the contrary, the *serial DDA combinator* creates a larger DDA by connecting two DDAs in a “sequential” fashion. New branches are added in the compound DDA that will connect a designated set of points from the first DDA to a designated set of points of the second DDA determined by a given transfer function.
3. The *sub-DDA combinator* is a unary operator resulting in a smaller DDA “forgetting” parts of the original DDA as specified in the construct. It resembles the sub-graph relation from graph-theory.
4. The *nesting DDA combinator* requires a *global DDA*, a collection of *local DDAs*, one for each global DDA point, and a family of transfer functions, see Fig. 1. Each point of the global DDA is replaced by its associated local DDA, and new dependency branches are added along the global depen-

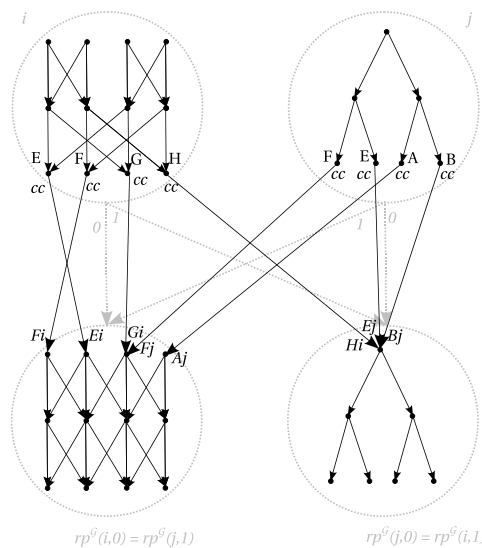


Figure 1: Detail of a nested DDA (black coloured), obtained using 4 points of a global DDA (grey coloured in the background), and the associated local DDAs.

dependencies between the local DDAs as specified by the transfer functions.

Combinators can be applied in arbitrary order when declaring compound DDAs. Implementing the combinators in the compiler according to the proposed formalisms ensures that the compound DDA is in effect a DDA, i.e., its components satisfy the axiomatic requirements of the DDA API which is quintessential in the framework.

We discuss the benefits of the combinators in the programming model context and instantiate their use by presenting a compound DDA-based polynomial multiplication.

## References

- [Ban96] Utpal K. Banerjee. *Dependence Analysis*. Kluwer Academic Publishers, Norwell, MA, USA, 1996. ISBN:0792398092.
- [BH09] Eva Burrows and Magne Haveræen. A hardware independent parallel programming model. *The Journal of Logic and Algebraic Programming*, 78(7):519 – 538, 2009. The 19th Nordic Workshop on Programming Theory (NWPT 2007). Available from: <http://dx.doi.org/10.1016/j.jlap.2009.06.002>.
- [BH12] Eva Burrows and Magne Haveræen. Programmable data dependencies and placements. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 31–40, New York, NY, USA, 2012. ACM. Available from: <http://doi.acm.org/10.1145/2103736.2103741>.
- [Bur14] Eva Burrows. Compiling a dataflow-based language abstraction onto an FPGA. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013, 10-13 September 2013, Garching (near Munich), Germany*, pages 507–514, 2014. Available from: <http://dx.doi.org/10.3233/978-1-61499-381-0-507>.
- [Hav00] Magne Haveræen. Efficient parallelisation of recursive problems using constructive recursion. In *Euro-Par 2000: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, number 1900 in Lecture Notes in Computer Science, pages 758–761, London, UK, 2000. Springer-Verlag. Available from: [http://dx.doi.org/10.1007/3-540-44520-X\\_104](http://dx.doi.org/10.1007/3-540-44520-X_104).
- [Sør98] Steinar Sørdeide. *Compiling Sapphire into Sequential and Parallel Code Using Assertions*. Master thesis, Department of Informatics, University of Bergen, Norway, P.O. Box 7800, N-5020 Bergen, Norway, 1998.

# Contract-based specification and verification of dataflow programs

Jonatan Wiik and Pontus Boström

Åbo Akademi University, Turku, Finland  
{jonatan.wiik, pontus.bostrom}@abo.fi

## 1 Introduction

Modern software systems are increasingly concurrent as the computational capacity of modern CPUs is improved mainly by increasing the number of processor cores. Writing software that efficiently exploits the capacity of such CPUs is hard. For this purpose new programming paradigms have been proposed. One such paradigm, which has gained a lot of attention within the signal processing domain is the dataflow paradigm. A dataflow program consists of a network of actors connected via asynchronous channels that describe the flow of data between actors. Each actor can execute concurrently when the required data is available on the incoming channels. As the only communication between actors is performed over channels, computations can easily be mapped to different processing units. In the general case dataflow programs have to be scheduled dynamically at runtime when run on general purpose hardware, which can cause significant runtime overhead. Consequently, different techniques to reduce the number of dynamic scheduling decisions have been investigated, e.g. [3].

In this work we present an approach to specification and automatic verification of dataflow programs based on assume-guarantee reasoning. The approach is based on annotating actors and networks with contracts stating functional properties which the actor or network should adhere to. The goal of the approach is to ensure functional correctness with respect to the contracts as well as deadlock freedom for dataflow networks. The contracts can potentially also be used to specify and prove properties which can be utilised to make scheduling decisions at compile-time. The work presented is a generalisation of previous work on verification of Simulink models [2], in which Simulink models are translated to synchronous data flow [5] (SDF) networks for verification. SDF is a subset of the the dataflow programs considered here, which can be statically scheduled.

## 2 Dataflow programs

We here consider dataflow programs in a language similar to the CAL actor language [4]. CAL has gained recent attention within the signal processing domain, and a subset of the language, named RVC-CAL, has also been standardised by ISO/IEC MPEG as part of the Reconfigurable Video Coding standard [6].

The dataflow programs considered here consist of a set of actors communicating over order-preserving channels of infinite size. Actors, which are allowed to have state, consist of a set of actions. The actor executes by firing an eligible action. An action is eligible depending on the number of tokens available on the incoming channels, the values of the tokens and the current state of the actor. Some examples of basic actors are listed in Fig. 1a. The actor *Add* has two input ports  $x1$  and  $x2$  and one output port  $y$ . The actor has one action, which reads one token from each of the input ports and outputs the sum of the read tokens. The actor *Delay*

<pre> <b>actor</b> Add() <b>int</b> x1, <b>int</b> x2 ==&gt; <b>int</b> y:   <b>action</b> x1:[i], x2:[j] ==&gt; y:[i+j] <b>end</b> <b>end</b>  <b>actor</b> Delay(<b>int</b> k) <b>int</b> x ==&gt; <b>int</b> y:   <b>initialize</b> ==&gt; y:[k] <b>end</b>   <b>action</b> x:[i] ==&gt; y:[i] <b>end</b> <b>end</b>  <b>actor</b> Split() <b>int</b> x ==&gt; <b>int</b> q, <b>int</b> u:   <b>action</b> x:[i] ==&gt; q:[i]     <b>guard</b> i &lt; 0   <b>end</b>   <b>action</b> x:[i] ==&gt; u:[i]     <b>guard</b> i &gt;= 0   <b>end</b> <b>end</b>  <b>actor</b> Sum() <b>int</b> x ==&gt; <b>int</b> y:   <b>inv</b> sum == (0::y)[last]    <b>int</b> sum := 0;    <b>action</b> x:[i] ==&gt; y:[sum]     sum := sum+i;   <b>end</b> <b>end</b> </pre>	<pre> <b>network</b> Sum() <b>int</b> x ==&gt; <b>int</b> y:   <b>entities</b>     a = Add();     d = Delay(0);   <b>end</b>   <b>structure</b>     x1: x --&gt; a.x1; x2: d.y --&gt; a.x2;     y: a.y --&gt; y; z: a.y --&gt; d.x;   <b>end</b>    <b>inv</b> delay(x2,1)   <b>inv</b> x2[next] == (0::y)[last]    <b>action</b> x:[i] ==&gt; y:[(0::y)[last]+i] <b>end</b>    <b>chinv</b> total(y) == read(x1)   <b>chinv</b> total(y) == read(x2)   <b>chinv</b> total(z) == read(x1)   <b>chinv</b> total(z) == read(x2)   <b>chinv</b> total(x2) == read(z)+1   <b>chinv</b> <b>forall</b> <b>int</b> i . 0 &lt;= i &amp;&amp; i &lt; total(y)     ==&gt; y[i] == x1[i]+x2[i]   <b>chinv</b> <b>forall</b> <b>int</b> i . 0 &lt;= i &amp;&amp; i &lt; total(z)     ==&gt; z[i] == x1[i]+x2[i]   <b>chinv</b> <b>forall</b> <b>int</b> i . 1 &lt;= i &amp;&amp; i &lt; total(x2)     ==&gt; x2[i] == z[i-1] <b>end</b> </pre>
(a)	(b)

Figure 1: (a) Implementations of some basic actors. (b) A dataflow network consisting of two basic actors of type *Add* and *Delay*.

delays the data on the input channel with one token. The actor has a special initialisation action outputting an initial token on the output port. This action is run only once when the actor is initialised. The actor *Split* is an example of a data-dependent actor, as its behaviour depends on the value of the incoming token. It outputs negative input tokens on port  $q$  and non-negative input tokens on port  $u$ . The actor *Sum* is an example of an actor with state. It accumulates the sum of the inputs it has received. A network consisting of one *Add* actor and one *Delay* actor is listed in Fig. 1b. It implements the same functionality as the actor *Sum* in Fig. 1a. It should be noted that the syntax used for specifying the network is not standard CAL. For instance RVC-CAL uses an XML-based format for networks. Networks in pure CAL do not have actions, but we use them here to describe the intended behaviour of the network. Hence a network action describes how the network should react when it receives input tokens.

### 3 Verification

For verification we encode the dataflow programs in the intermediate verification language Boogie [1]. An actor is verified by checking each action against its contract. Action contracts state preconditions and postconditions relating tokens on the input and output channels. For an



actor with invariant  $I$  and an action with precondition  $P$ , guard condition  $G$  and a postcondition  $Q$ , we need to check that  $Q$  and  $I$  hold after executing the action, assuming that  $P$ ,  $G$  and  $I$  hold before executing the action.

Verification of a network means checking that the network has the behaviour described by its network actions. To do this we need to express the relations between data on the channels in the network. We call these relations channel invariants. In the example in Fig. 1b, channel invariants are declared using the **chin** keyword. Channel invariants are required to hold during the execution of a network action, while network invariants, declared using the **inv** keyword, are required to hold before and after a network action is executed, but not necessarily while the action is executed. The channel invariants provided in the example in Fig. 1b express relations both between the number of tokens on the channels and between the data on the different channels. We for instance have the property that the total number of tokens (written as well as read) on the channel  $y$  during execution of the action is equal to the number of read tokens on channel  $x1$ . This type of properties are needed to ensure that the amount of tokens specified in the network action is available on the output channels after executing the action. Channel invariants also relate data on the different channels, for instance that the  $i$ :th token on channel  $y$  should be equal to the sum of the  $i$ :th tokens on channels  $x1$  and  $x2$ .

A network can be verified to be correct with respect to its contract in the following way: Assume that we have a network with network invariant  $I$  and an action with postcondition  $Q$ . Additionally assume that  $F_1, \dots, F_n$  are the firing conditions of each action  $A_1, \dots, A_n$  of every actor in the network and that  $C_1, \dots, C_m$  are the channel invariants of the network. Assuming that  $C_1, \dots, C_m$  hold, we check that  $C_1, \dots, C_m$  hold again after executing any actor  $A_i$  for which  $F_i$  evaluates to true. We additionally also check that the postcondition of the network action holds when no actor can be fired:  $\neg F_1 \wedge \dots \wedge \neg F_n \wedge C_1 \wedge \dots \wedge C_m \Rightarrow Q \wedge I$ .

## 4 Conclusions

We have outlined an approach to contract-based specification and verification of dataflow programs. The work is still in progress. To make the approach more usable in practice it would be important to infer as many as possible of the channel invariants for a network. We plan to investigate automatic inference of invariants for special classes of actor networks. We also plan to investigate the use of contracts to aid the compile-time scheduling of dataflow programs.

## References

- [1] M. Barnett, B.-Y. E. Chang, R. Deline, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO'05*, volume 4111 of *LNCS*. Springer, 2006.
- [2] P. Boström and J. Wiik. Contract-based verification of discrete-time multi-rate Simulink models. *Software & Systems Modeling*, 2015.
- [3] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silvén. Actor merging for dataflow process networks. *IEEE Transactions on Signal Processing*, 63(10):2496–2508, 2015.
- [4] J. Eker. and J. W. Janneck. CAL language report. Technical Report ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, 2003.
- [5] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1987.
- [6] M. Mattavelli, I. Amer, and M. Raulet. The reconfigurable video coding standard. *Signal Processing Magazine, IEEE*, 27(3), 2010.

# Tool Support for Component-Based Semantics

L. Thomas van Binsbergen<sup>1</sup>, Peter D. Mosses<sup>2</sup>, and Neil Sculthorpe<sup>2</sup>

<sup>1</sup> Department of Computer Science, Royal Holloway University of London, UK  
 ltvanbinsbergen@acm.org

<sup>2</sup> Department of Computer Science, Swansea University, UK  
 p.d.mosses@swansea.ac.uk, n.a.sculthorpe@swansea.ac.uk

## Abstract

The P<sub>L</sub>anCompS project has developed a component-based approach to formal semantics. Here, we present the tools we have implemented to support component-based language definitions, including semantics-based program execution. The talk includes a demonstration of the use of the tools.

## 1 Introduction and Background

The benefits of formal semantics are well known at NWPT. However, it requires a lot of work to produce a complete and accurate formal semantics for a major language; and when the language evolves, large-scale revision of the semantics may be needed to reflect the changes. The investment of effort needed to produce an initial definition, and subsequently to revise it, can discourage language developers from using formal semantics [3].

To improve the practicality of formal semantic definitions of larger languages, the P<sub>L</sub>anCompS project [9] proposes to base them on a collection of reusable components, and to implement tool support for their development and testing. Analogous practices are widely adopted in software engineering: developers rely on reusable components in the form of packages, and on IDEs when coding and testing.

**Component-based semantics.** In the P<sub>L</sub>anCompS approach, a reusable component of language definitions corresponds to a fundamental programming construct: a so-called ‘funcon’, which has a fixed operational interpretation. The formal semantics of each funcon is defined independently, using a modular variant of SOS [6, 7]. The collection of funcons is open-ended; crucially, adding new funcons never requires changes to the definition or use of previous funcons.

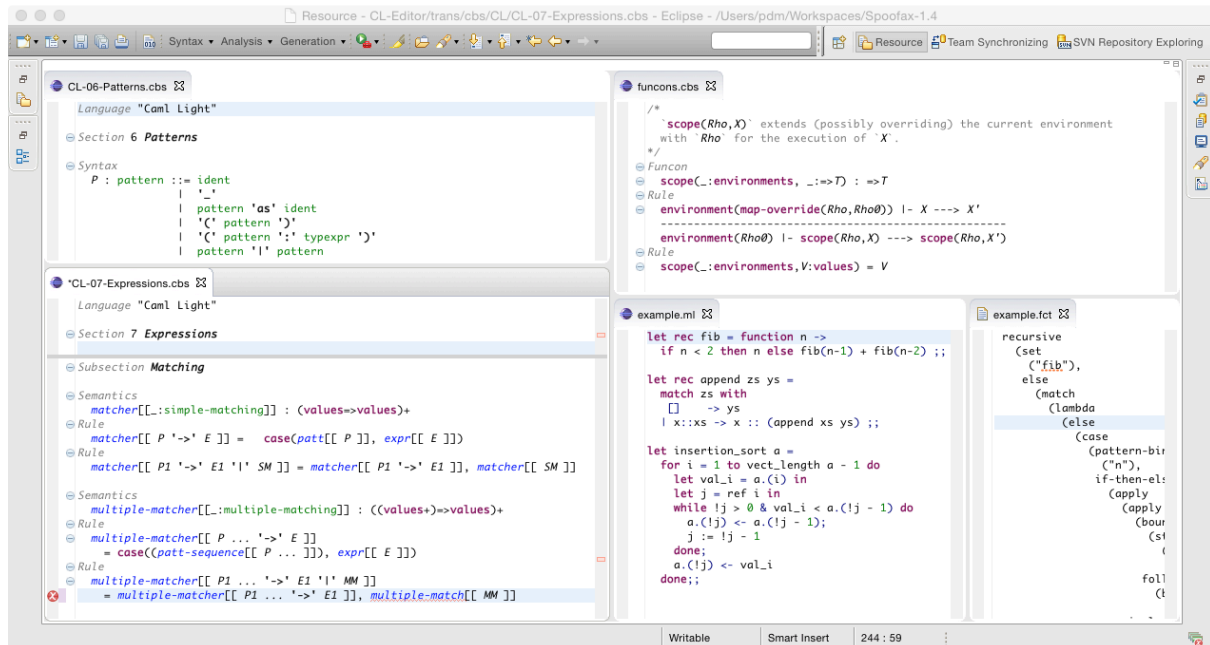
A component-based semantics of a programming language is defined by translating its constructs to funcons. The expectation is that many funcons can be widely reused in the definitions of different languages. An initial case study [1] gave a semantics for Caml Light [5] based on a preliminary collection of funcons; after completing a further case study (C<sup>#</sup>), the funcons used in the two language definitions are to be finalised and made freely available in a digital library.

**Contributions.** We introduce a unified meta-notation called CBS for defining abstract syntax of programming languages, translations from language constructs to funcons, and the semantics of the funcons themselves. To accompany CBS, we provide a complete tool chain for executing translation functions and running the resulting funcon terms. A further contribution is illustrating the usefulness of Spoofox [4] for generating IDEs to support semantic frameworks.

**Related work.** Other tools supporting development of semantic definitions and/or semantics-based program execution include the ASM tools, Ott, PLTRedex, the K tools, Maude, Melange, and DynSem. Some of the semantic frameworks supported by these tools have a high degree of modularity, but we are not aware of any that provide a collection of reusable components, apart from an exploratory definition of a modest collection of funcons in K [8].

## 2 Developing and Executing Language Definitions

We have implemented an IDE in Eclipse to support development and testing of component-based semantics. We have used Spoofox [4] to generate a CBS editor with many useful features, including syntax highlighting, syntax error recovery, hyperlinks from uses of symbols to their definitions, and flagging of undefined symbols.



The screenshot shows several files open during the development of the component-based semantics of Caml Light (CL). The top left pane is browsing the CL abstract syntax in the CBS definition of CL patterns. In the lower part of the bottom left pane, a CBS rule defining the translation of CL multiple matchings to funcons is being edited; the red mark in the margin flags an undeclared symbol. The colours and fonts distinguish the names of syntax nonterminals (green), funcons (red), semantic functions (blue italic), and variables (black italic).

Clicking on a name in a CBS editor shows its definition in a separate pane. The top right pane shows the definition of the operational semantics of the funcon for scoping declarations. The two panes on the lower right show a small CL test program and part of its translation to funcons. When the focus is on a CBS file specifying the semantics of CL, there is a button to (re)generate an executable translator from CL to funcons. While editing a CL program, the same button translates it to a funcon term (which can then be executed, see Sect. 3). On rebuilding the project, any open files with the results of translating test programs to funcons are updated accordingly. Entire test suites can be translated from a shell command line.

The implementation of the CBS editor in Spoofox involved writing an SDF3 grammar for the CBS language, some small files specifying the various editor services (highlighting, name resolution, menus, folding), and Stratego code to generate SDF3 grammars and Stratego rules from the ASTs of CBS specifications. Each semantic equation in CBS generates a corresponding Stratego rule, e.g.:

<pre> Rule   matcher[[ P '-&gt;' E ]] =     case(patt[[ P ]], expr[[ E ]])         </pre>	generates	<pre> to-funcons:   [[ matcher[: (P)-&gt;(E) :] ]   -&gt;   [[ case(patt[: (P) :], expr[: (E) :]) ]           </pre>
---	-----------	--

The generated SDF3 grammars provide the syntax for the semantic functions and metavariables that occur in the generated Stratego rules.

### 3 Executing Funcon Terms

We execute funcon terms using an interpreter written in Haskell. The interpreter provides an implementation of I-MSOS [7] specifications, the modular variant of SOS that CBS uses to specify funcons and semantic entities. The interpreter can be invoked from within Eclipse with its output printed to Eclipse's console .

The defining feature of I-MSOS is the implicit propagation of entities, and this is achieved in Haskell by using a monad in the implementation of the small-step evaluation function. The Haskell code corresponding to the CBS specifications of the individual funcons and semantic entities is systematically derived from the CBS rules. The use of a monad allows the resulting code to be as modular as I-MSOS rules: adding a new funcon or semantic entity requires no modification to the code for the existing funcons or semantic entities. Deriving the Haskell code is currently performed manually, although our aim is automate this process.

The CBS language includes a fixed universe of value types, and a set of operations on those types; these are provided by binding them to Haskell's data types and library functions. For nearly all cases, direct counterparts of the CBS value types and operations are available in the Haskell standard library.

Dynamic errors are handled gracefully by the interpreter, which reports the immediate cause of the error along with the current contents of the semantics entities and funcon term remaining to be executed. The interpreter also includes a parser and pretty printer for funcon terms, and an optional refocusing-based optimisation [2] that provides a more efficient evaluation strategy.

### 4 Conclusion

In a full version of this paper, we will explain how the CBS meta-notation supports modular specifications of funcons and semantic entities, and how these specifications can be translated to modular Haskell code. We will also explain how by categorising our semantic entities, we allow for the modular addition of new entities without concern for the order in which those entities are added, in contrast to a conventional approach using monad transformers.

### References

- [1] M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini. Reusable components of semantic specifications. In *Trans. AOSD XII*, volume 8989 of *LNCS*, pages 132–179. Springer, 2015.
- [2] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. BRICS Research Series RS-04-26, Department of Computer Science, Aarhus University, 2004. <http://www.brics.dk/RS/04/26/>.
- [3] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *HOP-L-III*, pages 12:1–12:55. ACM, 2007.
- [4] L. C. L. Kats and E. Visser. The Spoofox language workbench: Rules for declarative specification of languages and IDEs. In *OOPSLA '10*, pages 444–463. ACM, 2010.
- [5] X. Leroy. Caml Light manual, 1997. <http://caml.inria.fr/pub/docs/manual-caml-light>.
- [6] P. D. Mosses. Modular structural operational semantics. *J. LAP*, 60-61:195–228, 2004.
- [7] P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. In *SOS '08*, volume 229(4) of *ENTCS*, pages 49–66. Elsevier, 2009.
- [8] P. D. Mosses and F. Vesely. FunKons: Component-based semantics in K. In *WRLA '14*, volume 8663 of *LNCS*, pages 213–229. Springer, 2014.
- [9] PLANCOMPS: Programming language components and specifications. <http://www.plancomps.org>.

# A logical characterisation for input output conformance simulation $\text{iocos}_{\underline{}}$ \*

Luca Aceto, Ignacio Fábregas, Carlos Gregorio-Rodríguez, and Anna Ingólfssdóttir

Departamento Sistemas Informáticos y Computación  
Universidad Complutense de Madrid, Spain

ICE-TCS, School of Computer Science  
Reykjavik University, Iceland.

## 1 Introduction

Over the last couple of years we have been studying the input-output conformance simulation relation ( $\text{iocos}_{\underline{}}$ ) [5, 8, 6, 7] that refines the classic input-output conformance testing ( $\text{ioco}$ ) theory due to Tretmans. The work by Tretmans [10] has provided a widely used and theoretically well-founded framework for the Model Based Testing (MBT) community: it offers both offline and online testing algorithms [3], and there are several model-based test generation tools that implement the  $\text{ioco}$ -testing theory.

From a theoretical point of view, some interesting features of the  $\text{ioco}$ -framework are as follows: behaviours are modelled as labelled transition systems (LTS); quiescent states (see [9]) are considered; implementations should be input enabled; and the  $\text{ioco}$  relation is a trace-based semantics, and thus a linear semantics [11].

The  $\text{iocos}_{\underline{}}$  approach also considers LTS as models, quiescence, and shares much of the  $\text{ioco}$  philosophy, but it considers a wider domain of behaviours, not imposing, but allowing, implementations to be input enabled. The substantial difference between the two approaches is that the conformance relation underlying  $\text{iocos}_{\underline{}}$  is an input-output simulation (a branching-time semantics [11]) with greater discriminatory power than  $\text{ioco}$  (see [6, Theorem 1]).

Simulation is an important notion pervading many fields in computer science (model checking, concurrency theory, formal verification...), with a plethora of theoretical and practical applications. For example, results presented in [6], indicate that  $\text{iocos}_{\underline{}}$  may be used to minimise LTSs in model checking as a technique to alleviate the state explosion problem.

In more detail,  $\text{iocos}_{\underline{}}$  is a simulation-based semantics over LTSs developed under the assumption that systems have two kinds of transitions: input actions, those that the systems are willing to admit or respond to, and output actions, those produced by the system and that can be seen as responses or results. We call  $I$  the alphabet for input actions and write  $a?, b?, c? \dots$  for typical members of  $I$ . We denote with  $O$  the alphabet for output actions and use  $x!, y!, \delta! \dots$  to range over  $O$ .

A state with no output actions cannot autonomously proceed; such a state is called *quiescent*. For the sake of simplicity and without loss of generality (see for instance [10, 9]), we directly introduce the event of quiescence as a special action denoted by  $\delta! \in O$  into the definition of our models.

The formal definition of  $\text{iocos}_{\underline{}}$  considers the following functions on states of labelled transition systems:

---

\*Research partially supported by the Spanish MEC projects TIN2012-36812-C02-01 and TIN2012-39391-C04-04, the project 001-ABEL-CM-2013 within the NILS Science and Sustainability Programme and the project *Nominal SOS* (project nr. 141558-051) of the Icelandic Research Fund

$$\begin{aligned} \text{outs}(p) &= \{o! \mid o! \in O, p \xrightarrow{o!}\}, \text{ the set of initial outputs of a state } p. \\ \text{ins}(p) &= \{a? \mid a? \in I, p \xrightarrow{a?}\}, \text{ the set of initial inputs of a state } p. \end{aligned}$$

**Definition 1.** We say that a binary relation  $R$  of states in a labelled transition system is a  $\text{iocos}_{\subseteq}$ -relation if and only if for any  $(p, q) \in R$  the following conditions hold:

1.  $\text{ins}(q) \subseteq \text{ins}(p)$
2.  $\forall a? \in \text{ins}(q)$  if  $p \xrightarrow{a?} p'$  then  $\exists q'$  such that  $q \xrightarrow{a?} q' \wedge (p', q') \in R$ .
3.  $\forall o! \in \text{outs}(p)$  if  $p \xrightarrow{o!} p'$  then  $\exists q'$  such that  $q \xrightarrow{o!} q' \wedge (p', q') \in R$ .

We define the input-output conformance simulation ( $\text{iocos}_{\subseteq}$ ) as the union of all  $\text{iocos}_{\subseteq}$ -relations (the biggest  $\text{iocos}_{\subseteq}$ -relation). We will denote by  $\text{iocos}_{\equiv}$  the kernel of the  $\text{iocos}_{\subseteq}$  preorder.

## 2 Contribution: Logic for $\text{iocos}_{\subseteq}$

We present for the first time a logical characterization of the  $\text{iocos}_{\subseteq}$  relations, both the preorder and equivalence. This logic is a non-standard subset of Hennessy-Milner Logic and is rather *minimal* although convenient to characterize clearly the discriminating power of the  $\text{iocos}_{\subseteq}$  relation.

**Definition 2.** The syntax of the logic for  $\text{iocos}_{\subseteq}$ , denoted by  $\mathcal{L}_{\text{iocos}_{\subseteq}}$ , is defined by the following grammar.

$$\phi ::= \mathbf{tt} \mid \mathbf{ff} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a? \rangle \phi \mid \langle x! \rangle \phi,$$

where  $a? \in I$  and  $x! \in O$ . The semantics of the atomic propositions  $\mathbf{tt}$  and  $\mathbf{ff}$  and of the Boolean connectives  $\wedge$  and  $\vee$  is defined as usual. The modalities  $\langle a? \rangle$  and  $\langle x! \rangle$ , are defined as follows:

- $p \models \langle x! \rangle \phi$  iff  $p' \models \phi$  for some  $p \xrightarrow{x!} p'$ .
- $p \models \langle a? \rangle \phi$  iff  $p \xrightarrow{a?} \dashv$  or  $p' \models \phi$  for some  $p \xrightarrow{a?} p'$ .

It is well-known that every logic naturally induces a preorder on a given set of processes given by:  $p \leq_{\mathcal{L}} q$  iff  $\forall \phi \in \mathcal{L} \ p \models \phi$  then  $q \models \phi$ . Hence, the logic  $\mathcal{L}_{\text{iocos}_{\subseteq}}$  induces the preorder  $\leq_{\mathcal{L}_{\text{iocos}_{\subseteq}}}$ . The main contribution is that this logical preorder coincides with the  $\text{iocos}_{\subseteq}$  relation. That is, we have:

**Theorem 2.1** (Logical characterization for  $\text{iocos}_{\subseteq}$ ).  $p \text{ iocos}_{\subseteq} q$  iff  $\forall \phi \in \mathcal{L}_{\text{iocos}_{\subseteq}} \ p \models \phi$  then  $q \models \phi$ .

**Corollary 2.2.**  $p \text{ iocos}_{\equiv} q$  iff  $(\forall \phi \in \mathcal{L}_{\text{iocos}_{\subseteq}} \ p \models \phi \text{ iff } q \models \phi)$ .

**Corollary 2.3.** For all  $\phi$  in  $\mathcal{L}_{\text{iocos}_{\subseteq}}$  if we want to check  $p \models \phi$ , it is equivalent to minimise  $p$  to  $q$  (using the generalized coarsest partitioning algorithm from [12, 6] and decide whether  $q \models \phi$ ).

Finally, applying the results in [1] we can define the characteristic formula for each process in a finite LTS modulo the  $\text{iocos}_{\subseteq}$  preorder.

## 3 Future work

It seems natural to compare the previous logics for  $\text{iocos}_{\subseteq}$  with similar logics that are already in the literature. In particular, we find it interesting to explore its relation with the logics for ready simulation [11, 2], covariant-contravariant simulation and conformance simulation [4]. Since the study of property preservation for expressive logics is of great interest for the model checking community, we also plan to study how the properties preserved by  $\text{iocos}_{\subseteq}$  are related with those expressible in fragments of Action Based CTL and of the  $\mu$ -calculus.

## References

- [1] Luca Aceto, Anna Ingólfssdóttir, and Joshua Sack. Characteristic formulae for fixed-point semantics: A general framework. In Sibylle B. Fröschle and Daniele Gorla, editors, *Proceedings 16th International Workshop on Expressiveness in Concurrency, EXPRESS 2009, Bologna, Italy, 5th September 2009.*, volume 8 of *EPTCS*, pages 1–15, 2009.
- [2] David de Frutos-Escrig, Carlos Gregorio-Rodríguez, Miguel Palomino, and David Romero-Hernández. Unifying the linear time-branching time spectrum of process semantics. *Logical Methods in Computer Science*, 9(2:11):1–74, 2013.
- [3] René G. de Vries and Jan Tretmans. On-the-fly conformance testing using spin. *STTT*, 2(4):382–393, 2000.
- [4] Ignacio Fábregas, David de Frutos-Escrig, and Miguel Palomino. Logics for contravariant simulations. In *FMOODS-FORTE 2010*, volume 6117 of *Lecture Notes in Computer Science*, pages 224–231. Springer, 2010.
- [5] Carlos Gregorio-Rodríguez, Luis Llana, and Rafael Martínez-Torres. Input-output conformance simulation (iocos) for model based testing. In Dirk Beyer and Michele Boreale, editors, *FMOODS/FORTE*, volume 7892 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2013.
- [6] Carlos Gregorio-Rodríguez, Luis Llana, and Rafael Martínez-Torres. Effectiveness for input output conformance simulation iocos. In Erika Ábrahám and Catuscia Palamidessi, editors, *FORTE*, volume 8461 of *Lecture Notes in Computer Science*, pages 100–116. Springer, 2014.
- [7] Carlos Gregorio-Rodríguez, Luis Llana, and Rafael Martínez-Torres. Extending mcl2 with ready simulation and iocos input-output conformance simulation. In *SAC-SVT to appear*, 2015.
- [8] Luis Llana and Rafael Martínez-Torres. IOCO as a simulation. In Steve Counsell and Manuel Núñez, editors, *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers*, volume 8368 of *Lecture Notes in Computer Science*, pages 125–134. Springer, 2013.
- [9] Gerjan Stokkink, Mark Timmer, and Mariëlle Stoelinga. Talking quiescence: a rigorous theory that supports parallel composition, action hiding and determinisation. In Alexander K. Petrenko and Holger Schlingloff, editors, *MBT*, volume 80 of *EPTCS*, pages 73–87, 2012.
- [10] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.
- [11] Rob J. van Glabbeek. *Handbook of Process Algebra*, chapter The Linear Time – Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes, pages 3–99. Elsevier, 2001.
- [12] Rob J. van Glabbeek and Bas Ploeger. Correcting a space-efficient simulation algorithm. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 517–529. Springer, 2008.

# Using Typings as Types

Casper Bach Poulsen, Peter D. Mosses, and Neil Sculthorpe

Department of Computer Science, Swansea University, UK  
 casperbp@gmail.com, p.d.mosses@swansea.ac.uk, n.a.sculthorpe@swansea.ac.uk

## Abstract

In languages with dynamic scope, the free variables of an abstraction correspond to implicit parameters. The values of these parameters are determined by the bindings current where the abstraction is applied, but their allowed types can be checked statically. We give a novel type system for dynamic scope using types that involve type environments and intersections. We also explore how to give typing rules with a high degree of modularity using typings themselves as types.

## 1 Introduction and Background

Dynamic scope arises when the values of the free variables of a function body are those current when the function is applied, rather than when it was formed. Table 1 shows a conventional type system for a simply-typed  $\lambda$ -calculus with static scope (using notation following [4]). These rules are clearly unsound when the semantics requires dynamic scope.

In Sect. 2 we present a novel type system for a  $\lambda$ -calculus with dynamic scope. Here, abstractions with dynamic scope are first-class values: we let them be passed as arguments and returned by other abstractions, in contrast to previous type systems [1, 2, 6]. The required types include *typings*  $A \vdash \tau$  and intersections  $\sigma \wedge \tau$ .

The benefits of higher-order abstractions with dynamic scope in programming are debatable [6]. Our motivation for considering them comes from component-based semantics (CBS) [3, 7]. In this framework, the semantics of a programming language is defined by translating it to so-called fundamental programming constructs (‘funcons’). Functions with static scope are translated to funcon compositions of the form `close(thunk( $e$ ))`, where `close` takes an abstraction and returns its closure. The abstraction funcon `thunk( $e$ )` is a value incorporating an unevaluated expression  $e$ ; the free variables of  $e$  inherently have dynamic scope unless `close` is used.

CBS uses a modular variant of SOS for defining the dynamic semantics of each funcon independently [3]. In Sect. 3, we explore how to obtain modularity and scalability in type systems using richer forms of typings as types. As noted by Wells [8], typings in arbitrary type systems contain all the information from typing judgements other than the term being typed.

---


$$M, N ::= n \mid x \mid (M + N) \mid (\lambda x.M) \mid (M N)$$

$$\sigma, \tau ::= \text{int} \mid t \mid (\sigma \rightarrow \tau)$$

$$\emptyset \vdash n : \text{int} \quad (1)$$

$$\{x : \tau\} \vdash x : \tau \quad (2)$$

$$\frac{A \vdash M : \text{int} \quad A \vdash N : \text{int}}{A \vdash (M + N) : \text{int}} \quad (3)$$

$$\frac{A_x \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x.M) : (\sigma \rightarrow \tau)} \quad (4)$$

$$\frac{A \vdash M : (\sigma \rightarrow \tau) \quad A \vdash N : \sigma}{A \vdash (M N) : \tau} \quad (5)$$

$$\frac{A \vdash M : \tau}{A' \vdash M : \tau} \quad A \subseteq A' \quad (6)$$


---

Table 1: A conventional type system for static scope



## 2 A Type System for Dynamic Scope

When an abstraction  $\lambda x.M$  with dynamic scope is applied to an argument, the context should provide a bound value for each of its free variables. The types of these bound values, along with the type of the argument to be associated with  $x$ , should ensure that  $M$  is well-typed. We obtain types for abstractions with dynamic scope by enriching the types used in Table 1 with typings  $A \vdash \tau$ , indicating the requirement of a context providing variables according to the type environment  $A$ :

$$\sigma, \tau ::= \text{int} \mid t \mid (\sigma \rightarrow \tau) \mid (A \vdash \tau) \mid (\sigma \wedge \tau)$$

The type of an abstraction with dynamic scope is of the form  $A \vdash (\sigma \rightarrow \tau)$ , where  $\sigma$  and  $\tau$  may also involve type environments.

Rules (7–9) in Table 2 replace (4–6) in Table 1. The type environment  $A_x$  is as  $A$  except that any type constraint for  $x$  is ignored. Type environment union  $A_1 \cup A_2$  assumes  $A_1$  and  $A_2$  have disjoint domains, whereas intersection  $A_1 \wedge A_2$  combines the type constraints of  $A_1$  and  $A_2$  with intersection of the types of any common variables. The intersection  $A \wedge A'$  is needed in (8) because  $x$  may be required to have one type when evaluating  $M$  to an abstraction, and a different type when evaluating the body of the abstraction; similarly in (10).

---

$\frac{A \vdash M : \tau}{\emptyset \vdash \lambda x.M : (A_x \vdash (\sigma \rightarrow \tau))} \quad (A(x)=\sigma) \quad (7)$	$M ::= \dots \mid \text{close}(M)$
$\frac{A \vdash M : (A' \vdash (\sigma \rightarrow \tau)) \quad A \vdash N : \sigma}{(A \wedge A') \vdash (M N) : \tau} \quad (8)$	$\frac{A \vdash M : (A' \vdash (\sigma \rightarrow \tau))}{(A \wedge A') \vdash \text{close}(M) : (\emptyset \vdash (\sigma \rightarrow \tau))} \quad (10)$
$\frac{A \vdash M : \sigma}{A' \vdash M : \tau} \quad (A \vdash \sigma) <: (A' \vdash \tau) \quad (9)$	

---

Table 2: A type system for dynamic scope, and an extension with explicit closures

A typing rule corresponding to (4) can be derived for the term  $\text{close}(\lambda x.M)$  from (7) and (10), showing that such abstractions are supposed to have static scope (cp. [3, §3.4]).

The subtype relation on types (implicit in the use of  $\sigma \wedge \tau$ ) is also used on type environments and typings. Some of its properties are shown in Table 3.

---

$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{(\sigma \rightarrow \tau) <: (\sigma' \rightarrow \tau')} \quad (11)$	$(A_x \cup \{x : \tau\}) <: A_x \quad (13)$
$\frac{A' <: A \quad \sigma <: \tau}{(A \vdash \sigma) <: (A' \vdash \tau)} \quad (12)$	$\frac{A_x <: A'_x \quad \tau <: \tau'}{(A_x \cup \{x : \tau\}) <: (A'_x \cup \{x : \tau'\})} \quad (14)$

---

Table 3: Some subtyping rules

As an illustrative example, consider the dynamically scoped term

$$M = (\lambda f. (\lambda y. f 1) 2) (\lambda x. x + y)$$

The abstraction  $\lambda x. x + y$  has a free variable  $y$  that has to be bound to a value of type  $\text{int}$  when the abstraction gets applied, and we can derive  $\emptyset \vdash (\lambda x. x + y) : (\{y : \text{int}\} \vdash (\text{int} \rightarrow \text{int}))$ . The application  $(\lambda y. f 1) 2$  satisfies this constraint, and we can derive  $\emptyset \vdash M : \text{int}$ .

### 3 Modularity

For specifying a transition from  $M$  to  $M'$  in dynamic semantics, CBS provides the notation  $R \vdash \langle M, S \rangle \xrightarrow{W} \langle M', S' \rangle$  where  $R$  consists of read-only entities (e.g. environments  $\rho$ ),  $S$  and  $S'$  consist of mutable entities (e.g. stores  $\sigma$ ), and  $W$  consists of write-only entities (e.g. outputs  $\alpha$ ). Entities can be omitted, and are then implicitly propagated in transition rules. For instance,

$$\frac{\rho \vdash \langle M, \sigma \rangle \xrightarrow{\alpha} \langle M', \sigma' \rangle}{\rho \vdash \langle (M; N), \sigma \rangle \xrightarrow{\alpha} \langle (M'; N), \sigma' \rangle} \text{ can be abbreviated to } \frac{M \rightarrow M'}{(M; N) \rightarrow (M'; N)}$$

and  $\rho \vdash \langle ((); N), \sigma \rangle \dot{\rightarrow} \langle N, \sigma \rangle$  abbreviated to  $(((); N) \rightarrow N$ . Modular foundations for such abbreviations are provided by a rule-by-rule translation to MSOS (see [3]) where all auxiliary entities are incorporated in labels.

To obtain a similar degree of modularity for typing rules, we propose to let the types of terms be general typings that contain the corresponding types of all auxiliary entities. For example, consider  $R \vdash \langle \tau, S \rangle \xrightarrow{W} \langle \tau', S' \rangle$  where  $R$  may include the type environment ( $A$ ),  $S$  and  $S'$  may include store types ( $\Sigma$ ), and  $W$  may include interactive behaviour types ( $\alpha$ ). The type  $\tau$  is for explicit arguments of abstractions, and  $\tau'$  for computed values. Using implicit propagation for omitted entities, we could abbreviate

$$\frac{M : A \vdash \langle \tau, \Sigma_1 \rangle \xrightarrow{\alpha_1} \langle \text{int}, \Sigma_2 \rangle \quad N : A \vdash \langle \tau, \Sigma_2 \rangle \xrightarrow{\alpha_2} \langle \text{int}, \Sigma_3 \rangle}{(M + N) : A \vdash \langle \tau, \Sigma_1 \rangle \xrightarrow{\alpha_1 \cup \alpha_2} \langle \text{int}, \Sigma_3 \rangle} \text{ to } \frac{M : \Rightarrow \text{int} \quad N : \Rightarrow \text{int}}{(M + N) : \Rightarrow \text{int}}$$

and abbreviate  $n : \emptyset \vdash \langle \cdot, \Sigma \rangle \dot{\rightarrow} \langle \text{int}, \Sigma \rangle$  to  $n : \Rightarrow \text{int}$ .

### 4 Conclusion and Future Work

We have given a novel type system for a  $\lambda$ -calculus with dynamic scope, illustrated the typing of terms where abstractions with free variables are passed as arguments, and suggested a technique to obtain modularity in type systems. We now aim to prove the soundness of the type system, extend it with universal quantification and recursive types, and obtain principal typings [4, 5, 8].

### References

- [1] R. Chugh. A fix for dynamic scope. Unpublished abstract, presented at *ML '13*. Available from <http://people.cs.uchicago.edu/~rchugh/papers/>, 2013.
- [2] R. Chugh. IsoLATE: A type system for self-recursion. In *ESOP '15*, volume 9032 of *LNCS*, pages 257–282. Springer, 2015.
- [3] M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini. Reusable components of semantic specifications. In *Trans. AOSD XII*, volume 8989 of *LNCS*, pages 132–179. Springer, 2015.
- [4] T. Jim. What are principal typings and what are they good for? In *POPL '96*, pages 42–53. ACM, 1996.
- [5] T. Jim. A polar type system. In *ITRS '00, ICALP Satellite Workshops*, pages 323–338, 2000.
- [6] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: Dynamic scoping with static types. In *POPL '00*, pages 108–118. ACM, 2000.
- [7] PLANCOMPS: Programming language components and specifications. <http://www.plancomps.org>.
- [8] J. Wells. The essence of principal typings. In *ICALP '02*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.

## Appendix

*This appendix shows how we derived the typing claimed in Sect. 2, and sketches a denotational semantics for dynamic scope. It is not intended for inclusion in the final version.*

The claimed typing is:

$$\emptyset \vdash (\lambda f. (\lambda y. f \ 1) \ 2) (\lambda x. x + y) : \text{int} \quad (15)$$

For any  $A$  and  $\tau'$  we derive:

$$\{f : (A \vdash (\text{int} \rightarrow \tau'))\} \wedge A \vdash f \ 1 : \tau' \quad (16)$$

$$\emptyset \vdash (\lambda y. f \ 1) : ( (\{f : (A_y \cup \{y : \tau\} \vdash (\text{int} \rightarrow \tau'))\} \wedge A_y) \vdash (\tau \rightarrow \tau') ) \quad (17)$$

$$\{f : (A_y \cup \{y : \text{int}\} \vdash (\text{int} \rightarrow \tau'))\} \wedge A_y \vdash (\lambda y. f \ 1) \ 2 : \tau' \quad (18)$$

$$\emptyset \vdash \lambda f. (\lambda y. f \ 1) \ 2 : (A_{yf} \vdash ((A_y \cup \{y : \text{int}\} \vdash \text{int} \rightarrow \tau') \rightarrow \tau')) \quad (19)$$

Taking  $A = \{y : \text{int}\}$  gives:

$$\emptyset \vdash \lambda f. (\lambda y. f \ 1) \ 2 : (\emptyset \vdash (\{y : \text{int}\} \vdash \text{int} \rightarrow \tau') \rightarrow \tau') \quad (20)$$

We also have:

$$\emptyset \vdash (\lambda x. x + y) : (\{y : \text{int}\} \vdash (\text{int} \rightarrow \text{int})) \quad (21)$$

Taking  $\tau' = \text{int}$ , the claimed typing follows.

### A denotational semantics for dynamic scope

$$\begin{aligned} V &= Z + F \\ F &= Env \rightarrow V \rightarrow V \\ Env &= Var \rightarrow V \\ [[M]] &: Env \rightarrow V \\ [[n]]\rho &= n \\ [[x]]\rho &= \rho(x) \\ [[M + N]]\rho &= [[M]]\rho|_Z + [[N]]\rho|_Z \\ [[\lambda x.M]]\rho &= \lambda \rho'. \lambda v. [[M]](\rho'[x \mapsto v]) \\ [[M \ N]]\rho &= ([[M]]\rho|_F)(\rho)([[N]]\rho) \end{aligned}$$

# Towards user-friendly and efficient analysis with Alloy

Xiaoliang Wang, Adrian Rutle, and Yngve Lamo

Bergen University College,  
P.O. Box 7030, N-5020, Bergen, Norway  
Email: xwa, aru, yla@hib.no

In Model-driven engineering (MDE), structural models (also called static models in [6]) represent software at the early phases of software development. They identify the artifacts and their relationships in the problem domain. These models can be specified as graph-based structures and constraints in different formalisms, e.g., UML class diagram [8] and the Object Constraint Language (OCL) invariants [7]; in the structures, nodes represent the artifacts while edges represent the relationships; the constraints express requirements of the problem domain. An instance of a structural model is a graph which is well-typed by the underlying graph of the model, and, in addition, satisfies all the constraints of the model.

Usually, structural models are specified by a modelling language within a modelling tool; this may cause errors. Thus, these models should be verified to ensure correctness. In addition, in MDE, models are gradually refined in subsequent phases which then finally result in software. Therefore, the verification of models can avoid propagating of errors into the software. Moreover, it is obvious that finding design mistakes as early as in the modelling phase helps to build better software at a lower cost.

Different properties of structural models are studied in MDE [3]. For instance, *consistency* requests that a model has at least one instance; *lack of redundant constraints* requests that, given a model, there exists no constraint  $C_1$  that can be derived from another constraint  $C_2$ , i.e., there exists at least one instance of the model which satisfies  $C_1$  but not  $C_2$ . These properties can be categorised into *validity*, i.e., whether all the instances of a model satisfy a property, and *satisfiability*, i.e., whether there exists an instance which satisfies a property.

Several approaches have been presented to verify such properties on structural models [6]. Generally, they translate a structural model and a property into a specification in some formalism, e.g., Relational Logic [2, 4], etc. Then the specification is analysed by theorem provers or constraint solvers to answer whether the model satisfies the property. But these approaches are not integrated into the modelling tools; to use these approaches, the model designers have to switch from the modelling tool to a verification tool and need background in the verification methods. Moreover, most approaches present instances when the properties are satisfied, but give no feedback when the properties are violated. This is not convenient for model designing.

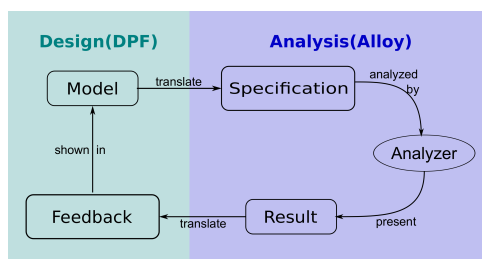


Figure 1: Workflow for analysing structural models using Alloy

In this work, we present a bounded verification approach of structural models using Alloy [1] and integrate it into a modelling tool DPF Model Editor [10]. The procedure of the approach is

illustrated in Fig. 1. It translates a structural model specified in Diagram Predicate Framework (DPF) [9] and a property into an Alloy specification. Then, the specification is examined by the Alloy Analyzer to check if the model satisfies the property or not. If the property is satisfied (violated), an instance (counterexample) of the model is generated. Otherwise, it means that there are some problems with the model. Then, the problematic part of the model will be highlighted and displayed in the DPF Model Editor to assist the model designer to identify the problem. For example, a civil status model which modifies the traditional civil status model in [5] originally specified in UML and OCL is present in Figure 2. It is inconsistent and the constraints which contradict each other are highlighted and presented in Figure 3. Thus, the model designer can verify the model under design and receive user-friendly feedback which he can understand, without knowing the underlying verification technique<sup>1</sup>.

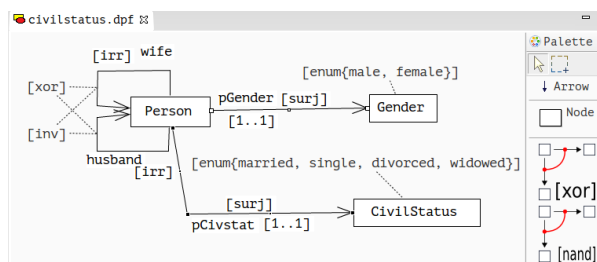


Figure 2: Civil Status Model in DPF

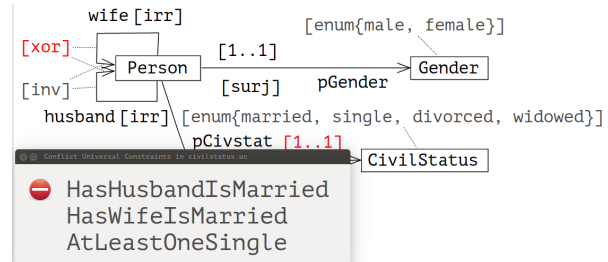


Figure 3: Highlight the problem

The approach is bounded; the approach finds instances or counterexamples which satisfies or violated properties within a bounded search space. The space is determined by a *scope*; i.e., a user-defined number which restricts the number of instances of each model element. However, there is no systematic way to decide which scope is needed, and, although different scopes could be sufficient for different model elements, the same size is usually used for all model elements. In addition, this approach has scalability problems since the search space grows exponentially along with the scope. It means that the verification of large models with a large search space may take long time or become intractable.

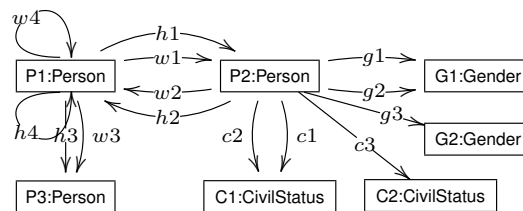


Figure 4: The scope graph for the civil status model

To solve these issues, we propose two contributions. The first one is to initialise a systematic way to decide the search space based on constraints (properties) definitions; here we focus on constraints which can be expressed in FOL. Given such constraints, we assume that there exists a *scope graph* such that, for each instance (counterexample) of the model, there exists smaller or equal instances (counterexamples) that are contained by the scope graph. However, since FOL is undecidable, such a scope graph do not exist for arbitrary constraints. In this work, we construct an approximation of the scope graph based on the syntax of the constraints. The

<sup>1</sup>This part which verifies structural models and presents the result of verification user-friendly, along with the splitting technique which is present in the sequel, is submitted to and accepted by Modevva2015.

approximation of the scope graph can be used to derive a scope to verify a structural model. For example, the approximation of the scope graph for the civil status model is shown in Figure 4. It contains 3 **Person**, 4 **husband**, 4 **wife**, 3 **pGender**, 3 **pCiviStat**, 2 **Gender**, 2 **CivilStatus**. This can be used as the scope for the consistence check of the model.

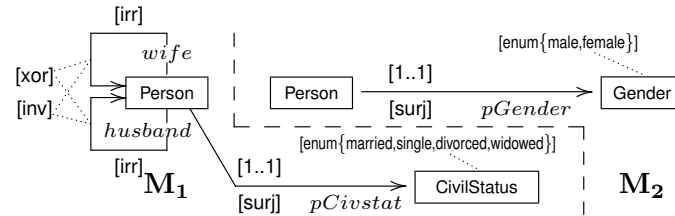


Figure 5: Submodels

The second contribution is a splitting technique for verification. A model can be split into submodels based on the *factors* of the constraints, i.e., the model elements which are affected by the constraints. We will look for submodels which are *left-total*, i.e. submodels of which the instances can be extended to instances of the model. We outline an approach to find these left-total submodels based on *forbidden patterns* of the constraints. That is, these submodels do not contain any match of patterns which violate (or are forbidden by) any constraints of the model. Then the validation of a model can be reduced to the validation of its left-total submodels. The civil status model can be split into two submodels which are shown in Figure 5. The submodel  $M_1$  is left-total and the consistence check of the model can be reduced to the consistence check of  $M_1$  rather than the whole model. An experimental result shows that it alleviates the scalability problem.

## References

- [1] Alloy. *Project Web Site*. <http://alloy.mit.edu/community/>.
- [2] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 2009.
- [3] Jordi Cabot, Robert Clarisó, and Daniel Riera. On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software*, 93:1–23, 2014.
- [4] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.
- [5] Martin Gogolla, Fabian Büttner, and Jordi Cabot. Initiating a benchmark for UML and OCL analysis tools. In *7th TAP*, pages 115–132, 2013.
- [6] Carlos A. González and Jordi Cabot. Formal verification of static software models in MDE: A systematic review. *Information & Software Technology*, 56(8):821–838, 2014.
- [7] Object Management Group. *Object Constraint Language Specification*, February 2014. <http://www.omg.org/spec/OCL/2.4/>.
- [8] Object Management Group. *Unified Modeling Language Specification*, May 2015. <http://www.omg.org/spec/UML/2.5/>.
- [9] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, University of Bergen, 2010.
- [10] Lamo Yngve, Wang Xiaoliang, Mantz Florian, Bech Øyvind, Sandven Anders, and Rutle Adrian. DPF Workbench: a multi-level language workbench for MDE. In *Proceedings of the Estonian Academy of Sciences*, pages 3–15, 2013.

# A Property Specification Language for Runtime Verification of Executable Models

Fernando Macias, Adrian Rutle, and Volker Stolz

Bergen University College  
`first.last@hib.no`

The definition of workflows is a complex task, comprising aspects such as time constraints, failure detection and recovery. Executable modelling is a promising concept for the simplification of workflow modelling, verification and execution. Hence, the verification of executable models, especially using runtime verification and monitoring, is required. We present a metamodelling approach to the combined modelling of workflows and the temporal properties used to define requirements and constraints on them. Here, a model at a certain level describes a modelling language which can be used to specify models in the level below. First, we give an overview of a generic workflow modelling framework for executable models. Then, we highlight the design and implementation of the property specification language. Both the workflows and their temporal properties are specified as graph-based structures. For the temporal properties, we draw on the well-known Linear Time Logic (LTL), which has already been used successfully in checking whether an (execution) path satisfies a given property [3]. A key point of the proposed language is its direct applicability on the running instances of the workflows, instead of the execution logs as it is usually done. Because of this, the atomic propositions of our language are graphs which are matched against the actual running instances while monitoring, and hence translating *match/no match* to *true/false* respectively. In our approach we exploit facilities from deep metamodelling [4] to establish links between the running instances of the workflows and the atomic propositions, using the *typing* relationship. Deep metamodelling enables us to define types for other levels in the hierarchy than the one directly below. In addition, we will borrow the concept of *linguistic extension* and *double typing* of model elements [4].

## Model execution and verification

This work is framed in a bigger proposal for the creation of a metamodelling framework which allows for the definition of executable modelling languages. To achieve this goal, we have outlined a deep metamodelling hierarchy where the two topmost levels are fixed (see Fig. 1). In them, the common elements for any executable language are defined. With these two levels as a starting point, the user can define her own modelling language using as many levels as required. The bottommost level will contain the running instances that the execution runtime can interpret and modify in every time step through model transformations. Note that the modelling levels  $M_2$  through  $M_4$  are displayed only to be used as examples and are inspired by the workflow modelling language defined in [6, 5], while  $M_0$  and  $M_1$  are fixed. The syntax of the language defined at level  $M_2$  is out of the scope of this paper; the interested reader may consult the references above. It is worth pointing out, however, the meaning of the dashed arrows: they represent the *typing* relationship, which indicates the type of the element according to the metamodel in the level above. Formally, this relationship is defined as a graph homomorphism from lower level models to upper level models. We can also indicate this relation with the colon notation “:”, e.g. `Task:Executable`. Note that in Fig. 1 we have used the concrete syntax for the models  $M_2$  through  $M_4$ , meaning that, e.g. a `Flow` instance, which should be a node, is actually represented as an arrow.

The same applies to the `[and]` and `[xor]` constraints in  $M_3$ . For the sake of clarity, not all the typing relationships are displayed. Next, we focus on the property definition language which is defined as a linguistic extension (i.e. a parallel metamodel) on the left-hand side. This technique consists of a separate model aside the vertical hierarchy, whose elements can be instantiated at any level of the hierarchy. In our work, this technique allows to apply such properties in any level using double typing. Further explanations of this can be found in the next section.

As a consequence of the choice of models for the representation of both the workflows and the temporal properties, we propose two disjoint sets of model transformations. The first one defines the semantics for the evolution of the running workflow instances in every time step (see [5, 2]). The second one defines the expansion of the temporal properties in the same stepwise manner, and determines how the properties are checked against the running workflow instances. This second set of model transformations is briefly introduced at the end of the next section. We discuss the design space of this construction and show its applicability to runtime verification and monitoring, where the properties are checked against the running system, as opposed to model checking, where the whole state space of the model is explored. In the case of the violation of a temporal property some actions could be taken, e.g. generating a warning.

## Property Specification Language

The language, as inspired by LTL, contains the the main temporal operators: **F** (eventually), **G** (always), **U** (Until), **R** (release) and **X** (next). It also contains the well-known boolean operators  $\neg$ ,  $\wedge$ ,  $\vee$  and  $\Rightarrow$  and the terminal symbols  $\top$  and  $\perp$ .

In order to create a consistent syntax for the language, all these operators inherit from one of the abstract classes `UnaryOperator`, `BinaryOperator` or `AtomicProposition` (see Fig. 1). All these, in turn, inherit from `Formula`, and can contain instances of `Formula` elements at the same time. This allows, in a grammar-like style, for the nesting of operators with the correct cardinality. Hence, a `Property` contains a single operator, which then contains the remaining operators in a tree structure; i.e. the models represent abstract syntax trees.

The two remaining elements in the language are `Model` and `Element`. In these elements lies the expressive power of our language: any element in any model in any level can be typed by `Element` (hence the redundancy) in addition to a type in their corresponding metamodel. One of the examples of this double typing shown in Fig. 1 is `Start crane operation`, which is both typed by `Task` and `Element`. This allows us to specify properties *on any* modelling level, or even *across* modelling levels, i.e. the same property can use `Model` instances which contain `Element` instances at different levels of the hierarchy. We call these *cross-level* properties.

In the sample hierarchy (levels  $M_2$ - $M_4$ ) in Fig. 1, properties specified using elements from  $M_3$  as well as properties connecting elements from  $M_3$  and  $M_4$  can be understood as requirements specifications. In general, this consideration can be applied to the second-to-last level, where

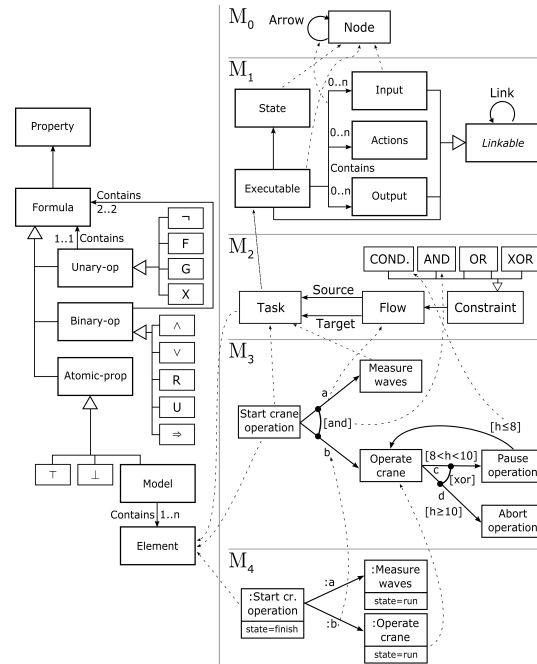


Figure 1: Global modelling hierarchy



the workflow itself is designed, right above the running instance. To illustrate our proposal, we show a possible temporal property *liveness* for the preceding workflow (Fig. 1). Note that this property is deliberately convoluted for the sake of exemplification.

Fig. 2 shows the model version of the liveness property which reads as follows in natural language: “For every  $X:Task$ , if we find a  $:X$  with the state *enable*, we should eventually find a  $:X$  with the state *run* and eventually after that a  $:X$  with the state *finish*”. The main highlights are (1) the expressiveness of the language, which allows for the definition of a strict requirement which includes the sequence of states that the element must go through; and more importantly, (2) the inclusion of a variable element,  $X:Task$ , of the level above the running instances,  $M_3$ .

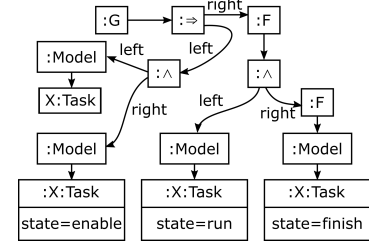


Figure 2: Liveness property

Finally, the semantics of the language is specified using model transformation rules. These rules check in a stepwise manner the specified properties against the running instance of the workflow. In order to do that, we use LTL expansion rules [ref], e.g.  $G(f) = f \wedge X(G(f))$ , for any formula  $f$ . This expansion is necessary to decompose the rules and extract atomic properties that can be checked in the current point in time in the workflow execution. Checking an atomic proposition in a workflow instance means finding a graph homomorphism (i.e. a match) from the underlying graph of the proposition to the underlying graph of the workflow instance, and hence translating *match/no match* to *true/false* respectively. We have already implemented a prototype with this semantics using EMF [1] and ATL [1].

## Conclusions and future work

We have presented a modelling language for the specification of temporal properties. While the idea of applying LTL monitoring to workflows is not new, cf. [8], our focus here is on the integration into a generic modelling framework for the specification and execution of workflows. We have also showed its two main characteristics: (1) the focus on the specification of temporal properties that have to be checked by means of runtime verification, instead of model checking; and (2) the flexibility it offers for the specification of *cross-level* properties. In this context, we plan to address in the immediate future what we call *multi-instance* properties, i.e. the ones that are checked against more than one workflow instance running in parallel. For this, it is required to extend the language with a means of quantification over instances [7].

## References

- [1] Eclipse Modeling Framework. *Web site*. <http://www.eclipse.org/modeling/emf>.
- [2] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *IJSTTT*, 14(1):15–40, February 2012.
- [3] M. Leucker et al. A brief account of runtime verification. *JLAP*, 78(5):293–303, 2009.
- [4] A. Rossini, J. De Lara, E. Guerra, A. Rutle, and U. Wolter. A formalisation of deep metamodelling. *Formal Aspects of Computing*, 26(6):1115–1152, 2014.
- [5] A. Rutle, W. MacCaull, H. Wang, and Y. Lamo. A metamodelling approach to behavioural modelling. In *BM-FA in conjunction with ECMFA*, Kgs. Lyngby, Denmark, July 2012. ACM.
- [6] A. Rutle, H. Wang, and W. MacCaull. A Formal Diagrammatic Approach to Compensable Workflow Modelling. In Z. Liu and A. Wassyng, editors, *FHIES*, volume 7789 of *LNCS*, pages 194–212. Springer, 2013.
- [7] V. Stolz. Temporal assertions with parametrized propositions. *JLC*, 20(3):743–757, 2010.
- [8] W. M. P. van der Aalst, H. T. de Beer, and B. F. van Dongen. Process mining and verification of properties: An approach based on temporal logic. *OTM’05*, pages 130–147. Springer, 2005.

# On endofunctors modelling higher-order behaviours

Marco Peressotti

Department of Mathematics and Computer Science, University of Udine, Italy  
marco.peressotti@uniud.it

It is well known that *higher-order systems*, i.e. systems which can pass around systems of the same kind, like the  $\lambda$ -calculus [1, 3], CHOCS [18], the higher-order  $\pi$ -calculus [14], HOcore [10], etc., are difficult to reason about. Many bisimulations and proof methods have been proposed also in recent works [4, 9, 11, 12, 15–17]. This effort points out that a definition of *abstract* higher-order behaviour is still elusive. In this work, we show how these abstract behaviours can be modeled as the *final coalgebras* of suitable *higher-order behavioural functors*.

Coalgebras are a well established framework for modelling and studying concurrent and reactive systems [13]. In this approach, we first define a *behavioural endofunctor*  $B$  over  $\mathbf{Set}$  (or other suitable category), modelling the computational aspect under scrutiny; for  $X$  a set of states,  $BX$  is the type of behaviours over  $A$ . Then, a system over  $A$  corresponds to a  $B$ -coalgebra, i.e. a map  $\alpha : X \rightarrow BX$  associating each state with its behaviour. The crucial step of this approach is defining the functor  $B$ , as it corresponds to specify the behaviours that the systems are meant to exhibit. Once we have defined a behavioural functor, many important properties and general results can be readily instantiated, such as the existence of the *final  $B$ -coalgebra* (containing all abstract behaviours), the definition of the canonical *coalgebraic bisimulation* (which is the abstract generalization of Milner’s strong bisimilarity) and its coincidence with behavioural equivalence [2], the construction of canonical *trace semantics* [8] and *weak bisimulations* [5], the notion of *abstract GSOS* [19], etc. We stress the fact that behavioural functors are *syntax agnostic*: they define the semantic behaviours, abstracting from any particular concrete representation of the systems.

Despite these results, a general coalgebraic treatment of higher-order systems is still missing. In fact, defining these functors for higher-order behaviours is challenging. In order to describe the problem, let us consider first a functor for representing the behaviour of a first-order calculus, like CCS with value passing:

$$B : \mathbf{Set} \rightarrow \mathbf{Set} \quad BX = \mathcal{P}_f(L \times V \times X + L \times X^V + X) \quad (1)$$

where  $L$  is the set of labels and  $V$  is the set of values. This functor is well-defined, and it admits a final coalgebra which we denote by  $\nu B$ ; the carrier of this coalgebra is the set of all possible behaviours, i.e., synchronization trees labelled with nothing, input or output actions. Now, in a higher-order calculus like HO $\pi$ , the values that processes can communicate are processes themselves. Semantically, this means that a higher-order behaviour can communicate *behaviours*; hence, in the definition (1) we should replace  $V$  with the carrier of  $\nu B$ , as follows:

$$B_{ho} : \mathbf{Set} \rightarrow \mathbf{Set} \quad B_{ho}X = \mathcal{P}_f(L \times |\nu B_{ho}| \times X + L \times X^{|\nu B_{ho}|} + X) \quad (2)$$

But this means that we are defining  $B_{ho}$  using its own final coalgebra  $\nu B_{ho}$ , which can be defined (if it exists) only after  $B_{ho}$  is defined—a circularity!

We think that this circularity is the gist of higher-order behaviours: any attempt to escape it would be restricting and distorting. One may be tempted to take as  $V$  some (syntactic) representation of behaviours (e.g., processes), but this would fall short. First, the resulting

behaviours would not be really higher-order, but rather behaviours manipulating some *ad hoc* representation of behaviours. Secondly, we would need some mechanism for moving between behaviours and their representations—which would hardly be complete. Third, the resulting functor would not be abstract and independent from the syntax of processes, thus hindering the possibility of reasoning about the computational aspect on its own, and comparing different models sharing the same kind of behaviour.

Endofunctors describing behaviours with input and outputs as (1) can be seen as endofunctors with *mixed-variance parameters*, e.g.  $F: \mathbf{Set}^{op} \times \mathbf{Set} \rightarrow [\mathbf{Set}, \mathbf{Set}]$  where  $F(A, B) = Id^A + B$ . Since we are interested in endofunctors with a final coalgebra we shall consider “schemes of endofunctors” in some suitable subcategory  $\mathbf{E}$  of  $[\mathbf{C}, \mathbf{C}]$ . Actually, parameters do not have to be in  $\mathbf{C}$ , for instance, the functor  $Id^A: \mathbf{Top} \rightarrow \mathbf{Top}$  can be seen as parametric in the exponentiable space  $A \in \mathbf{ExpTop}$ . In this situation, we need some coherent way back to the category of parameters i.e. a functor from  $\mathbf{E}$  to  $\mathbf{D}$ . An example of such situation is offered by taking  $\mathbf{E}$  to be the category of biconinuous endofunctors over  $\mathbf{Set}$ , as shown in [?]: every such functor admits a final coalgebra whose carrier set is endowed with a complete partial order naturally induced by the final sequence. Therefore, given:

$$F: \mathbf{D}^{op} \times \mathbf{D} \rightarrow \mathbf{E} \quad \text{and} \quad N: \mathbf{E} \rightarrow \mathbf{D}$$

(the driving example being  $N = |\nu - |$ ) we are interested in finding  $B: \mathbf{C} \rightarrow \mathbf{C} \in \mathbf{E}$  s.t.:

$$B \cong F(NB, NB)$$

or, equivalently,  $Z \in \mathbf{C}$  s.t.:

$$Z \cong NF(Z, Z)$$

since  $B \cong F(Z, Z)$  and  $Z \cong NB$ .

A  $\omega\mathbf{Cat}$ -category is a 2-category whose hom-categories have colimits for all  $\omega$ -chains and composition preserves them. A  $\omega\mathbf{Cat}_0$ -category is a  $\omega\mathbf{Cat}$ -category whose hom-categories have initial objects and composition preserves them. Any category enriched over  $\mathbf{Cpo}$ , the category of continuous maps between complete partial orders, such as  $\mathbf{Cpo}$  itself is an  $\omega\mathbf{Cat}$ -category. Likewise, any category enriched over  $\mathbf{Cpo}_\perp$  is an  $\omega\mathbf{Cat}_0$ -category.

**Theorem 1.** *Assume  $\mathbf{D}$  and  $\mathbf{E}$  to be  $\omega\mathbf{Cat}_0$ -categories with pseudo initial objects and pseudo colimits of  $\omega$ -chains of coreflections. Assume  $F$  and  $N$  to be a pseudo  $\omega\mathbf{Cat}$ -functors. There exist  $G: \mathbf{C} \rightarrow \mathbf{C} \in \mathbf{E}$  and  $Z \in \mathbf{C}$  as above.*

*Example 2 (Higher-order deterministic processes).* Let  $\mathbf{C}$  and  $\mathbf{D}$  be  $\mathbf{Cpo}_\perp$  and consider the parametric family of endofunctors  $Id^A + Id + B$ . The components of the coproduct model inputs, internal moves, and termination with outputs, respectively. In the higher-order version of this behaviour inputs and outputs are behaviours of the same kind. If we set aside for a moment syntax and binders (which can be modelled in suitable presheaf categories [6, 7]), this behaviour offers an operational semantics for the  $\lambda$ -calculus: intuitively, inputs are transitions  $t \xrightarrow{z} tz$  whereas internal reductions and outputs are transitions  $(\lambda x.t)z \rightarrow t[x/z]$ .

The functor  $F(A, B) = Id^A + Id + B$  is  $\mathbf{Cpo}$ -enriched and each endofunctors in its image has a final coalgebra in  $\mathbf{Cpo}_\perp$ . By restricting to the image of  $F$ , the assignment  $|\nu - |$  defines a  $\mathbf{Cpo}$ -enriched functor  $N: \mathbf{E} \rightarrow \mathbf{Cpo}_\perp$ . Thus, we have  $B_\lambda: \mathbf{Cpo}_\perp \rightarrow \mathbf{Cpo}_\perp$  and  $Z_\lambda \in \mathbf{Cpo}_\perp$  s.t.:

$$B_\lambda \cong Id^{Z_\lambda} + Id + Z_\lambda \quad \text{and} \quad Z_\lambda \cong |\nu B_\lambda|.$$

A  $B_\lambda$ -coalgebra  $(X \rightarrow X^{Z_\lambda} + X + Z_\lambda)$  is a strict continuous map assigning to each state of its carrier (a) a strict continuous function assigning a continuation to any value in input, (b) or a new state (internal step), (c) or a value (output). Indeed values are elements of the  $\mathbf{CPO}_\perp$   $Z_\lambda$  carrying the final coalgebra of  $B_\lambda$  i.e. behaviours for  $B_\lambda$  itself.

**Finite-order approximations** It might not be so easy to work with the solution  $B \cong F(N^{\text{op}}(B), N(B))$  since it is defined in terms of its own final coalgebra. The reason of this is rooted in the inherent circularity of higher-order definitions; circularity that resurfaces in the definition of  $B$  and many related constructions such as  $B$ -bisimulations.

The steps in the computation of the fixed point  $B$  are finite-order behaviours  $B_n$  (and  $Z_n$ ):

$$Z_0 = 0_{\text{D}} \quad Z_n = N(B_n) \quad B_0 = 0_{\text{E}} \quad B_{n+1} = F(Z_n, Z_n)$$

approximating  $B$  (for it is the (co)limit of the resulting  $\omega$ -chain of coreflections in  $\mathbf{E}$ ). Therefore,  $B$ -bisimulations may be given by induction on  $n$  deriving  $B_{n+1}$ -bisimulations from projections to  $n$ -order behaviours and  $B_n$ -bisimulations. Embeddings guarantee coherence.

## References

- [1] S. Abramsky. The lazy lambda calculus. *Research topics in functional programming*, pages 65–116, 1990.
- [2] P. Aczel and N. Mendler. A final coalgebra theorem. In D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Proc. CTCS*, volume 389 of *Lecture Notes in Computer Science*, pages 357–365, 1989. Springer.
- [3] H. Barendregt. *The lambda calculus: its syntax and its semantics*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.
- [4] L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012.
- [5] T. Brengos, M. Miculan, and M. Peressotti. Behavioural equivalences for coalgebras with unobservable moves. *CoRR*, abs/1411.0090, 2014.
- [6] M. Fiore and D. Turi. Semantics of name and value passing. In H. Mairson, editor, *Proc. 16th LICS*, pages 93–104, Boston, USA, 2001. IEEE Computer Society Press.
- [7] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In G. Longo, editor, *Proc. 14th LICS*, pages 193–202, 1999. IEEE Computer Society Press.
- [8] I. Hasuo, B. Jacobs, and A. Sokolova. Generic trace semantics via coinduction. *Logical Methods in Computer Science*, 3(4), 2007.
- [9] V. Koutavas, P. Blain Levy, and E. Sumii. From applicative to environmental bisimulation. In *Proc. MFPS, Electronic Notes in Theoretical Computer Science*, 276(0):215–235, 2011.
- [10] I. Lanese, J. A. Pérez, D. Sangiorgi, and A. Schmitt. On the expressiveness and decidability of higher-order process calculi. In *Proc. LICS*, pages 145–155. IEEE Computer Society, 2008.
- [11] S. Lenglet, A. Schmitt, and J. B. Stefani. Characterizing contextual equivalence in calculi with passivation. *Information and Computation*, 209(11):1390–1433, 2011.
- [12] A. Piérard and E. Sumii. A higher-order distributed calculus with name creation. In *Proc. LICS*, pages 531–540. IEEE Computer Society, 2012.
- [13] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.
- [14] D. Sangiorgi. Bisimulation for higher-order process calculi. *Information and Computation*, 131(2):141–178, 1996.
- [15] D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. In *Proc. LICS*, pages 293–302. IEEE Computer Society, 2007.
- [16] K. Støvring and S. B. Lassen. A complete, co-inductive syntactic theory of sequential control and state. In *Semantics and algebraic specification*, pages 329–375. Springer, 2009.

- [17] E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. *Journal of the ACM (JACM)*, 54(5):26, 2007.
- [18] B. Thomsen. Plain CHOCS: a second generation calculus for higher order processes. *Acta informatica*, 30(1):1–59, 1993.
- [19] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Proc. LICS*, pages 280–291. IEEE Computer Society Press, 1997.

# Open memory transactions in Haskell

Marino Miculan, Marco Peressotti and Andrea Toneguzzo

MADS, Department of Mathematics and Computer Science, University of Udine, Italy  
 marino.miculan@uniud.it, marco.peressotti@uniud.it, andrea.toneguzzo@spes.uniud.it

*Transactional memory* (TM) has emerged as a promising mechanism to replace locks [1,3]. The basic idea is to mark blocks of code as *atomic*; then, execution of each block will appear either if it was executed instantaneously, at some unique point in time, or, if aborted, as if it did not execute at all. This is obtained by means of *optimistic* executions: the blocks are allowed to run concurrently, and eventually if an interference is detected a transaction is restarted and its effects are rolled back. Differently from lock-based concurrency control mechanisms, transactions are composable and ensure absence of deadlocks and priority inversions, automatic roll-back on exceptions, and increased concurrency. Moreover, each transaction can be viewed in isolation as a *single-threaded* computation, significantly reducing programmer's burden.

However, in multi-threaded programming different transactions may need to interact and exchange data *before* reaching the commit phase. In this situation, transaction isolation is a severe shortcoming. A simple example is a synchronization (rendezvous) between threads belonging to different transactions. A naive attempt would be to use two semaphores `c1`, `c2` as shown aside. Unfortunately, this solution does not work: any admissible execution requires an interleaved scheduling between the two transactions, thus violating isolation; hence, the transactions deadlock as none of them can progress. It is important to notice that this deadlock arises because synchronization occurs between threads in *different* transactions; in fact, the solution above works for threads *outside* transactions, or within the *same* transaction.

In order to overcome this limitation, we propose a programming model for *safe, data-driven* interactions between memory transactions. The key observation is that *atomicity* and *isolation* should be seen as two disjoint computational aspects:

- an *atomic non-isolated* block of code is executed “all-or-nothing”, but its execution can overlap that of others and *uncontrolled* access to shared data is allowed;
- an *isolated* block of code is intended to be executed “as it were the only one” (i.e., in mutual exclusion with other threads), but no rollback on errors/exceptions is provided.

Thus, a “normal” block of code is neither atomic nor isolated; a mutex block (like Java *synchronized* methods) is isolated but not atomic; and a usual transaction is a block which is both atomic and isolated. Our claim is that atomic non-isolated blocks can be fruitfully used for implementing safe composable interacting transactions, henceforth called *open transactions*.

In this model, a transaction is composed by several threads, called *participants*, which can cooperate on shared data. A transaction commits when all its participants commit, and aborts if any thread aborts. Threads participating to different transactions *can* access to shared data, but when this happens the transactions are transparently *merged* into a single one. For instance, the two transactions of the rendezvous example above would automatically merge becoming the same transaction, hence the two threads can synchronize and proceed. Thus, this model relaxes the isolation requirement still guaranteeing atomicity and consistency; moreover, it allows for *loosely-coupled* interactions since transaction merging is driven only by run-time accesses to shared data, without any explicit coordination among the participants beforehand.

Open memory transactions in Haskell

Miculan, Peressotti, Toneguzzo

```

type ITM a
type OTM a
-- t is a placeholder for ITM or OTM --

-- Sequencing, do notation -----
(>>=) :: t a -> (a -> t b) -> t b
return :: a -> t a

-- Atomic and isolated computations ---
atomic  :: OTM a -> IO a
isolated :: ITM a -> OTM a
retry   :: ITM a
orElse  :: ITM a -> ITM a -> ITM a

-- Exceptions -----
throw :: Exception e => e -> t a
catch :: Exception e => t a ->
      (e -> t a) -> t a

-- Threading -----
fork :: OTM () -> OTM ThreadId

-- Transactional memory -----
data OTVar a
newOTVar  :: a -> ITM (OTVar a)
readOTVar :: OTVar a -> ITM a
writeOTVar :: OTVar a -> a -> OTM ()

```

Figure 1: The base interface of OTM.

In the rest of this abstract we gradually present the primitives from the OTM library (showed in Figure 1) by some illustrative examples. The discussion is meant to be introductory and informal. The formal semantics has been omitted due to space constraints but it is based on the calculus we presented in [2], with minor variations to accommodate the richer types of OTM. This library has been implemented in Haskell using the standard STM library.

Suppose we want to delegate some long task to another thread and then collect the result once it is ready. An intuitive way to achieve this is by means of *futures*, i.e. “proxy results” that will be produced by the worker threads.

A future can be implemented in OTM as a *transactional variable* `OTVar` holding a value of type `Maybe a` i.e. a type that is “not-ready-yet” (`Nothing`) or actually holds something of type `a` (e.g. “Just 42”):

```
type Future a = OTVar (Maybe a)
```

To access the promised value a call to `getFuture f` should block if the value is not ready yet. In OTM (and also STM) blocking a thread translates into “this thread has been scheduled too early” and the scheduler is informed of this fact by `getFuture :: Future a -> ITM a` means of the primitive `retry`. Therefore we can `getFuture f = case (readOTVar f) of` implement `getFuture` as `aside`. Note that there is `Nothing -> retry` no point in blindly restarting the transaction until `Just val -> return val` at least one of the transactional variables has been modified. Instead an implementation will use the information contained in the transaction log (which is needed by the optimistic execution strategy) to watch for `f` to change. The above snippet is executed in isolation to guarantee consistency between reading and testing the case (actually, it is also an implementation in STM).

Transaction openness comes into play when a worker is spawn. STM does not allow for thread creation, thus forcing us to implement `spawn` in IO. This is indeed possible, but forking and creating the future cannot be guaranteed to be atomic, let alone the creation of several workers. For instance, being able to create workers and futures inside an open transaction allows us to propagate exceptions and abort to all workers. Therefore we can implement `spawn` as follows:

```

spawn :: OTM a -> OTM (Future a)    worker :: Future a -> OTM ()
spawn work = do                    worker v = do
  future <- newTVar Nothing          res <- work
  fork (worker future)              writeOTVar (Just v) $! res
  return future

```

Because of its type, spawning multiple computations inside the same transaction is as simple

as function composition:

```
spawnMany :: [OTM a] -> OTM [Future a]  getAllFutures :: [Future a] -> ITM [a]
spawnMany = mapM spawn                    getAllFutures = mapM getFuture
```

While programming with OTM we should prefer the strictest kind of transactions otherwise we will lose some information because of a less precise type. For instance, consider the following alternative implementations of `getAllFutures`:

```
get' = isolated . (mapM getFuture)      get'' = mapM (isolated getFuture)
```

Although both have type `[Future a] -> OTM [a]`, the first is executed in isolation (like `getFutures`) whereas the second allows workers to proceed during the traversal of the list. Same type, different degree of concurrency.

Transactions can be composed as *alternatives* thanks to the primitive `orElse` which firstly attempts to execute its first argument as a sub-transaction and its second whenever the first one retries. The following function collects the value that is made available first:

```
getAnyFuture (f:fs) = (getFuture f) 'orElse' (getAnyFuture fs)
```

**Example: Petri nets** Petri nets might be easily implemented in OTM: places are transactional variables holding a number of tokens (`OTVar Peano`). Tokens can be added and removed with the latter operation being blocking.

```
data Place = OTVar Peano

take :: Place -> ITM ()
take var = do
  t <- redOTVar var
  case t of
    Zero -> retry
    Succ v -> writeOTVar var v

newPlace :: Peano -> ITM Place
newPlace = newOTVar

put :: Place -> ITM ()
put var = do
  v <- readOTVar var
  writeOTVar var (Succ v)
```

Transitions are IO threads that repeatedly consume tokens from their input places and produce tokens to their output places, atomically:

```
transition :: [Place] -> [Place] -> IO ThreadId
transition ins outs = forkIO (forever fire)
  where
    fire :: IO ()
    fire = atomic $ do
      (isolated take) 'all' ins
      (isolated put) 'all' outs
    all :: (a -> OTM b) -> [a] -> OTM ()
    all f = mapM_ f
```

Although each transition fires sequentially, the firing of different transitions happens in a true concurrent way since transactions are open and isolation is limited to each `take/put` operation.

## References

- [1] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA*, pages 289–300. ACM, 1993.
- [2] M. Miculan, M. Peressotti, and A. Toneguzzo. Open transactions on shared memory. In *Proc. COORDINATION*, pages 213–229, 2015.
- [3] N. Shavit, D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.



# Extending a Theorem Prover for Deductive Program Verification

Evgenii Kotelnikov<sup>1</sup>, Laura Kovács<sup>1</sup>, and Andrei Voronkov<sup>2</sup>

<sup>1</sup> Chalmers University of Technology, Gothenburg, Sweden  
evgenyk@chalmers.se, laura.kovacs@chalmers.se

<sup>2</sup> The University of Manchester, Manchester, UK  
andrei@voronkov.com

## 1 Introduction

Automated program analysis and verification requires discovering and proving program properties. Typical examples of such properties are loop invariants or Craig interpolants. These properties usually are expressed in combined theories of various data structures, such as integers and arrays, and hence require reasoning with both theories and quantifiers. Recent approaches in interpolation and loop invariant generation [9, 7, 4] present initial results of using first-order theorem provers for generating quantified program properties. First-order theorem provers can also be used to generate program properties with quantifier alternations [7]; such properties could not be generated fully automatically by any previously known method. Using first-order theorem prover to generate, and not only prove program properties, opens new directions in analysis and verification of real-life programs.

First-order theorem provers, such as iProver [5], E [10], and Vampire [8], lack however various features that are crucial for program analysis. For example, first-order theorem provers do not yet efficiently handle (combinations of) theories; nevertheless, sound but incomplete theory axiomatisations can be used in a first-order prover even for theories having no finite axiomatisation. Another difficulty in modelling properties arising in program analysis using theorem provers is the gap between the semantics of expressions used in programming languages and expressiveness of the logic used by the theorem prover. For example, a standard way to capture assignment in program analysis is to use a `let-in` expression, which introduces a local binding of a variable or a function, to a value. There is no local binding expression in first-order logic, which means that any modelling of imperative programs using first-order theorem provers at the backend, should implement a translation of `let-in` expressions.

Efficiency of reasoning-based program analysis largely depends on how programs are translated into a collection of logical formulas capturing the program semantics. The boolean structure of a program property that can be efficiently treated by a theorem prover is however very sensitive to the architecture of the reasoning engine of the prover. Deriving and expressing program properties in the “right” format therefore requires solid knowledge about how theorem provers work and are implemented — something that a user of a verification tool might not have. Moreover, it can be hard to efficiently reason about certain classes of program properties, unless special inference rules and heuristics are added to the theorem prover, see e.g. [3] when it comes to prove properties of data collections with extensionality axioms.

In order to increase the expressiveness of program properties generated by reasoning-based program analysis, the language of logical formulas accepted by a theorem prover needs to be extended with constructs of programming languages. This way, a straightforward translation of programs into first-order logic can be achieved, thus relieving users from designing translations which can be efficiently treated by the theorem prover. One example of such an extension is

recently added to the TPTP language [11] of first-order theorem provers, resembling `if-then-else` and `let-in` expressions that are common in programming languages. Namely, special functions `$ite_t` and `$ite_f` can respectively be used to express a conditional statement on the level of logical terms and formulas, and `$let_tt`, `$let_tf`, `$let_ff` and `$let_ft` can be used to express local variable bindings for all four possible combinations of logical terms (`t`) and formulas (`f`). While satisfiability modulo theory (SMT) solvers integrate `if-then-else` and `let-in` expressions, in the first-order theorem proving community so far only Vampire supports such expressions.

We aim for facilitation of reasoning-based program analysis by developing new theories for first-order theorem provers and extending them with features of programming languages. Our recent result in this direction is formalisation of a first-order logic with first class boolean sort that can be efficiently treated by a theorem prover and allows for simpler translation of some program properties, compared to an ordinary many-sorted first-order logic. Our current work is an implementation of this logic in Vampire. Both the theoretical result and the progress on the implementation are summarised in the following section.

## 2 First Class Boolean Sort

Our recent work [6] presented a modification of many-sorted first-order logic that implements a first class boolean sort. We called this logic FOOL, standing for first-order logic (FOL) + boolean sort. FOOL extends ordinary many-sorted FOL with (i) the boolean sort such that terms of this sort are indistinguishable from formulas and (ii) `if-else-else` and `let-in` expressions. FOOL formulas can be translated to ordinary FOL formulas while preserving models and can hence be treated by a first-order theorem prover.

FOOL is more expressive than FOL. Function and predicate symbols in FOOL can take boolean arguments, formulas in FOOL can be quantified over boolean variables and occur as arguments when the sorts coincide.

We argue that these extensions are useful in reasoning about problems coming from program analysis. A boolean expression in a programming language can be treated both as a value (for example, in a form of a boolean flag, passed as an argument to a function) and as a formula (for example, a loop condition). A translation of a program property into FOOL would not distinguish these cases, whereas a translation into an ordinary FOL would. In the later case, a possible solution would be to map the boolean type of programs to a user-defined boolean sort, postulate axioms about its semantics, and manually convert boolean terms into formulas where needed. This approach however is cumbersome and possibly inefficient due to the way a theorem prover might treat one of the boolean theory axioms.

In contrast to that, for a translation from FOOL to FOL given in [6] we described a modification of superposition calculus that can reason efficiently about the FOL formulas obtained with the translation. This modification involves replacement of the problematic theory axiom by an extra inference rule.

At this point we have an initial implementation of FOOL in Vampire that closely follows [6]. We were able to test the implementation on a set of 490 properties about (co)algebraic data-types, generated by the Isabelle proof assistant. All of these properties feature quantification over booleans and `if-then-else` expressions and were not previously directly expressible in TPTP. The preliminary results of our experiments show that native implementation of FOOL shows better performance than a naive translation of the boolean type described earlier. However, more experiments are still needed in order to make a conclusion.

Implementation of FOOL in Vampire allowed us to simplify the syntax of monomorphically

sorted TPTP, called TFF0 [12]. The lack of distinction between boolean terms and formulas made the distinction between `$ite_t` and `$ite_f` obsolete. They were replaced by a single `$ite` expression. Similarly, four variations of `let-in` expression were replaced by a single `$let`.

### 3 Future Work

Whereas we proposed a technique for efficient treatment of the boolean sort by a superposition-based theorem prover, no efficient translation of `if-then-else` and `let-in` expressions has been presented. This is left for future work.

Treatment of boolean terms as formulas was implemented in some other logics used in the automated deduction community. The core language of SMT-LIB [1], the collection of benchmarks for SMT-solvers, is a language of first-order logic with this property. The language of higher-order logic supported by theorem provers such as Isabelle is another example. FOOL is a novel result in the area of first-order reasoning and it bridges the semantic gap between logics. Particularly, FOOL is the smallest superset of the core language of SMT-LIB and monomorphically typed subset of TPTP. It means that a first-order theorem prover that supports FOOL can understand SMT-LIB problems without a special translation and we are planning to conduct experiments in reasoning about the corpus of SMT-LIB problems in Vampire.

FOOL is monomorphically typed. It is not hard however to extend it to a polymorphic case. TPTP defines a language of many-sorted FOL with rank-1 polymorphism called TFF1 [2], and implementation of a combination of FOOL and TFF1 is an interesting future work.

### References

- [1] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at [www.SMT-LIB.org](http://www.SMT-LIB.org).
- [2] J. C. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In *Proc. of CADE-24*, pages 414–420. Springer, 2013.
- [3] A. Gupta, L. Kovács, B. Kragl, and A. Voronkov. Extensionality Crisis and Proving Identity. In *Proc. of ATVA*, pages 185–200, 2014.
- [4] K. Hoder, L. Kovács, and A. Voronkov. Playing in the grey area of proofs. In *Proc. of POPL*, pages 259–272, 2012.
- [5] K. Korovin. iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *Proc. of IJCAR*, pages 292–298, 2008.
- [6] E. Kotelnikov, L. Kovács, and A. Voronkov. A First Class Boolean Sort in First-Order Theorem Proving and TPTP. In *CICM*, pages 71–86, 2015.
- [7] L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE*, pages 470–485, 2009.
- [8] L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In *Proc. of CAV*, pages 1–35, 2013.
- [9] K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS*, pages 413–427, 2008.
- [10] S. Schulz. System Description: E 1.8. In *Proc. of LPAR*, pages 735–743, 2013.
- [11] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.
- [12] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP Typed First-Order Form with Arithmetic. In *LPAR*, pages 406–419, 2012.

# Session-Based Compositional Verification on Actor-based Concurrent Systems \*

Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen

Department of Computer Science, Technische Universität Darmstadt, Germany  
 eduard.kamburjan@gmx.de, crystald@cs.tu-darmstadt.de, tc.chen@dsp.tu-darmstadt.de

**Motivations** Concurrent and distributed systems are the pillars of modern IT infrastructures. It is of great importance that such systems work properly. However, quality assurance of such systems is non-trivial since they depend on unpredictable factors, such as different processing speeds of independent components. Besides, it is non-trivial to ensure communication (composed by interactions) safety: as developers implement applications locally, the sending application's expected sequence of interactions may not fit the receiving application's. Thus interactions between endpoints could become inconsistent and unexpected messages may damage endpoint applications such that extra cost is caused to the overall system. These challenges motivate compositional frameworks combining precise modeling and analysis with suitable tool support for such kind of systems. In particular, it is crucial to provide a verification framework which is able to analyze the overall interactions and structurally verify endpoint applications in an intuitive way with respect to endpoint behavior.

Object orientation is the leading framework for concurrent and distributed systems. Concurrent objects combine object-orientation with the *actor* model [7]. Actors communicate with one another by asynchronous message passing, which allows the caller to continue with its own activity without blocking while waiting for the reply. Moreover, the notion of *futures* [6] improves this paradigm by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing future identities, the caller enables other objects to wait for the same method results.

**The Proposed Framework** In this work, we take *ABS* [5] as a modeling language for concurrent and distributed systems. *ABS* is based on Creol [10]. It is imperative, object-oriented, executable and it supports concurrent objects and shared futures.

The observable behavior of a system can be described by *communication histories* over observable events [8]. Due to asynchronous message passing in *ABS*, [3, 4] propose a disjoint event semantics, in which events are separated for method invocation, reacting upon a method call, resolving a future, and for fetching the value from a future. Each event is observable to only one object, which is the one that generates the event.

The theorem prover KeY-ABS [2] based on KeY [1] is developed for verifying history-based class invariants for *ABS* models. The class invariants can (1) relate the internal object states with the interactions between the current object and the surrounding environment, or (2) express the structure of histories local to the current object. The proof rule for compositional reasoning about *ABS* programs is given and proved sound in [4], by which system invariants can be obtained from the class invariants proved by KeY-ABS through history composition. This bottom-up verification approach by KeY-ABS is based on relay-guarantee mechanism for each class in the model. In this work, we propose a top-down verification approach to verify the overall behavior between concurrent and distributed endpoints.

---

\*The work has been supported by the EU project FP7-610582 *Envisage: Engineering Virtualized Services* (<http://www.envisage-project.eu>).

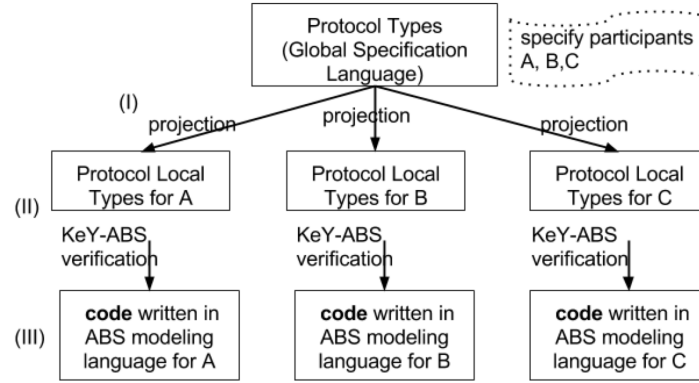


Figure 1: Session-Based Compositional Verification Framework

*Session types* [9] establish a means of typing concurrent and asynchronous interactions among distributed components. In this work, we propose a session-based verification framework for concurrent and distributed *ABS* models. We type applications' behaviors, which include the usage of *future*, with respect to sessions where the applications are participating in, and partition those behaviors based on sessions. We call the extended session types as *protocol types*, which not only enjoy all features defined in session types, but also specify the timing for invoking feature. Let  $p, q, ..$  range over endpoint process identifiers, let  $l$  be labels for options in a branching, and let  $f$  be future identities. The syntax of protocol types is defined below:

$$\begin{aligned}
(\text{Sorts}) \quad S &::= T \mid \langle G \rangle \\
(\text{Data types}) \quad T &::= \text{unit} \mid \text{bool} \mid \text{int} \mid \text{string} \mid \text{Fut}\langle T \rangle \\
(\text{Protocol Types}) \quad G &::= p \xrightarrow{f_j} q : \{l_j(T_j).G_j\}_{j \in J} \mid \text{Rel}(f) \mid G \parallel G \mid t \mid \mu t. G \mid \text{end} \\
(\text{Protocol Local Types}) \quad L &::= q!_{f_j} \{l_j(T_j).L_j\}_{j \in J} \mid p?_{f_j} \{l_j(T_j).L_j\}_{j \in J} \mid \text{Rel}(f) \mid t \mid \mu t. L \mid \text{end}
\end{aligned}$$

Sorts, denoted by  $S$ , range over data types, and  $\langle G \rangle$ , a closed protocol type (i.e. having no type variable) of a content. This implies that we can deliver a behavior (typed by  $G$ ) from one endpoint to another. Data types,  $T$ , include standard value types and future types,  $\text{Fut}\langle T \rangle$ , for the types of method parameters and method return results. A protocol type  $p \xrightarrow{f_j} q : \{l_j(T_j).G_j\}_{j \in J}$  globally describes an interaction behavior in which an endpoint process  $p$  sends a content of type  $T_k$  to another endpoint process  $q$ , where  $f_j$  is a future identity and label  $l_k \in \{l_j\}_{j \in J}$ ,  $k \in J = \{1..n\}$ . Then the global behavior continues with  $G_k$ . If  $J = \emptyset$ , then conventionally we write  $p \xrightarrow{f_j} q : (T).G$  to represent a simple sending and receiving interaction. In the protocol *local* types,  $q!_{f_j} \{l_j(T_j).L_j\}_{j \in J}$  and  $p?_{f_j} \{l_j(T_j).L_j\}_{j \in J}$  are corresponding to the interacting endpoints' behaviors defined in  $p \xrightarrow{f_j} q : \{l_j(T_j).G_j\}_{j \in J}$ . The former types the sender's behavior to send a message to  $q$ , while the later types the receiver's behavior to receive a message from  $p$ . The new type  $\text{Rel}(f)$  captures the process release point upon waiting for the future  $f$  to be resolved, i.e. containing method results.

For example, we can write

$$p_1 \xrightarrow{f_1} q_1 : \{ \text{BigD}(\text{string}).\text{Rel}(f_1).p_2 \xrightarrow{f_2} q_1 : (\text{unit}).\text{end}, \text{SmallD}(\text{bool}).q_1 \xrightarrow{f_1} p_1 : (\text{int}).\text{end} \}$$

to globally describe the following interactional features between  $p_1$ ,  $q_1$ , and  $p_2$ : there are two branches for the first interaction  $p_1 \xrightarrow{f_1} q_1$ . Branch *BigD* leads to compute a very big data at  $q_1$

(requested by  $p_1$ ), while branch *SmallD* is for computing a small data.  $q_1 \xrightarrow{f_1} p_1 : (int)$  implies that  $p_1$  will need to wait for the response from  $q_1$  before proceeding to the next action **end** (i.e. terminate). Since it is just a small data,  $p_1$  should not wait for a long time. However, for a very big data,  $p_1$  may waste lots of time for the response. Thus type  $\text{Rel}(f_1)$  is used in branch *BigD* to specify that  $p_1$  encounters a process release point and can proceed other actions while waiting for  $f_1$  to be resolved. It further specifies that, only when future  $f_1$  has been resolved, the whole global behavior can go to the next interaction  $p_2 \xrightarrow{f_2} q_1 : (unit)$ . We use  $G \parallel G$  for parallel composition, and  $t$  for type variable, and  $\mu t. G$  for a recursive type, where every  $t$  in the recursion body  $G$  is guarded by prefixes (i.e. contractive). Other terms for local types can be similarly explained.

**Concluding Remarks and Future Works** The main contributions in this work include (1) protocol types are extended by adding terms suitable for capturing the notion of *futures*, (2) the communication between different *ABS* endpoints, grouped by sessions, can be captured in protocol types and verified by the corresponding session-based composition verification framework, and (3) the local protocol types, projected from protocol types, of each endpoints can be translated and reformulated into history-based class invariants for KeY-ABS, see Figure 1. Based on this achievement, we also expect to extend the verification framework for *ABS* exception handling [11].

## References

- [1] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
- [2] C. C. Din, R. Bubel, and R. Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195 of *LNCS*, pages 517–526. Springer International Publishing, 2015.
- [3] C. C. Din and O. Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *JLAMP*, 83(5–6):360–383, 2014.
- [4] C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.
- [5] R. Hähnle. The abstract behavioral specification language: A tutorial introduction. In *Formal Methods for Components and Objects*, volume 7866 of *LNCS*, pages 1–37. Springer, 2013.
- [6] R. H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, Oct. 1985.
- [7] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI’73*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
- [8] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.
- [9] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, Jan. 2008.
- [10] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
- [11] I. Lanese, M. Lienhardt, M. Bravetti, E. Johnsen, R. Schlatte, V. Stolz, and G. Zavattaro. Fault model design space for cooperative concurrency. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, volume 8803 of *LNCS*, pages 22–36. Springer Berlin Heidelberg, 2014.

# StaRVOOrS: Unifying Static and Runtime Verification of Java

Wolfgang Ahrendt<sup>1</sup>, Jesús Mauricio Chimento<sup>1</sup>, Gordon Pace<sup>2</sup> and Gerardo Schneider<sup>3</sup>

<sup>1</sup> Chalmers University of Technology, Sweden.  
ahrendt@chalmers.se, chimento@chalmers.se

<sup>2</sup> University of Malta, Malta.  
gordon.pace@um.edu.mt

<sup>3</sup> University of Gothenburg, Sweden.  
gerardo@cse.gu.se

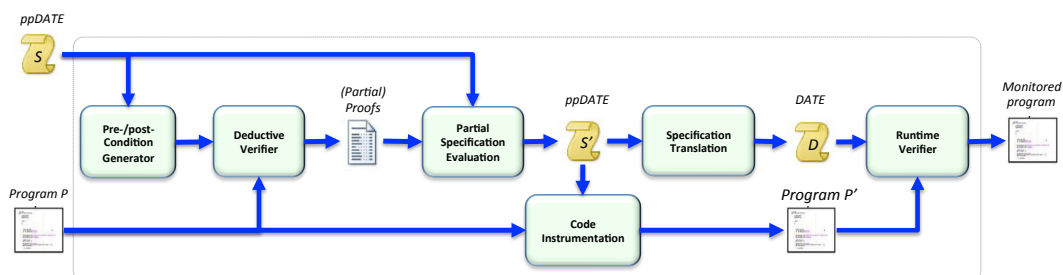
## 1 Introduction

Over the past decades, several forms of automated verification techniques have been proposed and explored in the literature. These techniques mostly fall in one of two categories: static and dynamic verification. *Runtime verification* is a dynamic technique concerned with the monitoring of software, providing guarantees that observed runs comply with specified properties. It is strong in analysing systems of a complexity that is difficult to address by *static verification*, like systems with numerous interacting sub-units, heavy usage of mainstream libraries, real (as opposed to abstract) data, and real world deployments. On the other hand, the major drawbacks of runtime verification are the impossibility to extrapolate correct observations to all possible executions, and that monitoring introduces runtime overheads. In the work we present here, these issues are addressed by combining runtime verification with static verification, such that: (i) Static verification attempts to ‘resolve’ those parts of the properties which can be confirmed statically; (ii) the static results, even if only partial, are used to change the property specification such that generated monitors will not check dynamically what was confirmed statically.

In addition to combining static and runtime verification, we introduce the specification language *ppDATE* [3], and a verification tool which embodies this approach, called STARVOORS [7], which captures both *control-oriented* properties (like the *DATE* language used in the runtime verification tool LARVA [9]) and *data-oriented* properties (like the *Java Modelling Language* JML [6, 10]).

## 2 The STARVOORS Framework

Below we show an abstract view of the framework, which was initially sketched in [4].



Given a Java program  $P$  and a specification  $\pi$  of the properties to be verified (given in the language *ppDATE*, see Sec. 3), these are transformed into suitable input for the deductive verifier KeY [5] which attempts to statically verify the properties related to pre/post-conditions. What is not proved statically will then be left to be proven at runtime. Here, not only the completed but also the *partial* proofs will be used by the *Partial Specification Evaluator* module in order to rewrite the original  $\pi$  into  $\pi'$  which triggers runtime checks for the parts which were not possible to prove statically. To achieve this, the original pre-conditions from  $\pi$  are refined to express also path conditions for not statically verified executions.

The *ppDATE* specification  $\pi'$  is then translated into a specification  $D$ , written in the *DATE* formalism [8] — a formalism suitable for the runtime verifier LARVA [9]. As *DATE* does not handle pre/post-conditions, these are simulated by pure *DATE* concepts. This also requires changes to the code base (done by the *Code Instrumentation* module), like adding counters to distinguish different executions of the same code unit, or adding methods which operationalise pre/post-condition evaluation. The instrumented program  $P'$  and the *DATE* specification  $D$  are passed on to the runtime verification tool LARVA, which uses aspect-oriented programming techniques to capture relevant system events and monitors, thus producing a monitored program, essentially equivalent to running the original program in parallel with a monitor of the original property, albeit more efficiently.

### 3 A Specification Language for Static and Runtime Verification of Data and Control Properties

STARVOORS uses *ppDATE* as the input language for its properties, which enables the combination of data- and control-based properties in a single formalism. *ppDATE*s are a composition of the control-flow language *DATE*, and of data-oriented specifications in the form of Hoare triples with pre-/post-conditions expressed using JML boolean expression syntax [10], which is designed to be easily usable by Java programmers. The data-oriented features of the specification appear in the states. A state may have a number of Hoare triples assigned to it. Intuitively, if Hoare triple  $\{\pi\}f\{\pi'\}$  appears in state  $q$ , the property ensures that: if the system enters code block  $f$  while the monitor lies in state  $q$  and precondition  $\pi$  holds, upon reaching the corresponding exit from  $f$ , postcondition  $\pi'$  should hold. To ensure efficient execution of monitors, *ppDATE*s are assumed to be deterministic by giving an ordering in which transitions are executed.

For a full and detail description of *ppDATE*, refer to [3].

### 4 STARVOORS Tool

The tool is a fully automated implementation of the theoretical results presented in [3, 4]. Given a property specification and the original Java program, our tool chain produces a statically optimised monitor and the weaved Java program to be monitored. This includes the automated triggering of numerous verification attempts of the underlying static verification tool, the analyses of resulting partial proofs, and the monitor generation.

The tool works following these steps: (1) A property is written using our script language for *ppDATE*; (2) Hoare triples are extracted from the specification of the property, are translated into JML contracts to be added to the Java files; (3) KeY attempts to verify all JML contracts, generating (partial) proofs, the analysis of which results in an XML file; (4) The *ppDATE* is refined based on the XML file; (5) Declarative pre/post-conditions are operationalised; (6) The code is instrumented with auxiliary information for the runtime verifier; (7) The *ppDATE*



specification is encoded into *DATES*; (8) The LARVA compiler generates a runtime monitor. See [7] for more details about the tool.

## 5 Conclusion

In [3] we have formalised the language for combining (partial) static and (optimised) runtime verification, which we called *ppDATE*, we have introduced an algorithm to transform a *ppDATE* specification in a *DATE* specification (the input language of the runtime verifier LARVA). In [7] we have introduced the fully automated tool STARVOORS, which implements the theoretical results presented in [3, 4]. StaRVOOrS combines the deductive theorem prover KeY and the runtime verification tool LARVA, and uses properties written using the *ppDATE* specification language. In addition, we have demonstrated the effectiveness of the tool by applying it to *Mondex* [1], an electronic purse application for smart cards products.

At the moment we are working on the proof of soundness of the *ppDATE* transformation algorithm introduced in [3] and we are analysing a new and larger case study based on *SoftSlate* [2], a full-featured, high-performance, open-source Java shopping cart that powers various of e-commerce websites.

Our future work includes: (i) improving the automation of the ‘operationalisation’ of pre/post-conditions containing algorithmic content; (ii) introduction of a mechanism to deal with the runtime verification of private information; (iii) development of an analyser for the output produced by the monitors generated by STARVOORS.

## References

- [1] MasterCard International Inc. Mondex. [www.mondexusa.com](http://www.mondexusa.com).
- [2] SoftSlate. [www.softslate.com](http://www.softslate.com).
- [3] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. A specification language for static and runtime verification of data and control properties. In *FM'15*, volume 9109 of *LNCS*, pages 108–125. Springer, 2015.
- [4] Wolfgang Ahrendt, Gordon Pace, and Gerardo Schneider. A Unified Approach for Static and Runtime Verification: Framework and Applications. In *ISOLA'12*, LNCS 7609, pages 312–326. 2012.
- [5] Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
- [6] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *SERP'02*, pages 322–328. CSREA Press, 2002.
- [7] Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon Pace, and Gerardo Schneider. StaRVOOrS: A tool for combined static and runtime verification of Java. In *RV'15*, LNCS. Springer, 2015. To appear. Available online at <http://www.cse.chalmers.se/~chimento/starvoors/publications.html>.
- [8] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In *FMICS'08*, volume 5596 of *LNCS*, pages 135–149. Springer-Verlag, September 2009.
- [9] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA - A Tool for Runtime Monitoring of Java Programs. In *SEFM'09*, pages 33–37. IEEE Computer Society, 2009.
- [10] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual. Draft 1.200*, 2007.

# A formal model for direct-style asynchronous observables

Philipp Haller<sup>1</sup> and Heather Miller<sup>2</sup>

<sup>1</sup> KTH Royal Institute of Technology, Sweden

phaller@kth.se

<sup>2</sup> EPFL, Switzerland

heather.miller@epfl.ch

## 1 Introduction

Asynchronous programming has been a challenge for a long time. A multitude of programming models have been proposed that aim to simplify the task. Interestingly, there are elements of a convergence arising, at least with respect to the basic building blocks: futures and promises have begun to play an increasingly important role in a number of languages like Java, C++, ECMAScript, and Scala. The Async extensions of F# [9], C# [1], and Scala [4] provide language support for programming with futures in *direct style*, by avoiding an inversion of control that is inherent in designs based on callbacks.

In this paper we present an integration of the Async model with a richer underlying abstraction, the *asynchronous observables* of the Reactive Extensions model [8]. An asynchronous observable is a stream of *observable events* which an arbitrary number of *observers* can subscribe to. The set of possible event patterns of asynchronous observables is strictly greater than those of futures. An observable (or stream) can (a) produce zero or more regular events, (b) complete normally, or (c) complete with an error (it is even possible for a stream to never complete.) Given the richer substrate of observables, the Async model has to be generalized in several dimensions.

We call our model RAY, inspired by its main constructs, *reactive async*, *await*, and *yield*. This paper makes the following contributions:

- The design of a new programming model, RAY, which integrates the Async model and the Reactive Extensions model;
- Structural operational semantics of the proposed programming model. Our operational semantics generalizes the formal model presented in [1] for C#'s *async/await* to asynchronous observables.

## 2 Background

**Scala Async.** Scala Async provides constructs that aim to facilitate programming with asynchronous events in Scala. The introduced constructs are inspired by extensions that have been introduced in C# version 5 [5]. The goal is to enable expressing asynchronous code in *direct style*, *i.e.*, in a familiar blocking style where suspending operations look as if they were blocking while at the same time using efficient non-blocking APIs under the hood. Example:

```
val respFut = async {
  val dayOfYear = await(futureDOY).body
  val daysLeft = await(futureDaysLeft).body
  Ok("" + dayOfYear + ": " + daysLeft + " days left!")
}
```

The `await` on line 2 causes the execution of the `async` block to suspend until `futureDOY` is completed (with a successful result or with an exception). When the future is completed successfully, its result is bound to the `dayOfYear` local variable, and the execution of the `async` block is resumed. When the future is completed with an exception (*e.g.*, because of a timeout), the invocation of `await` re-throws the exception that the future was completed with. In turn, this completes future `respFut` with the same exception. Likewise, the `await` on line 3 suspends the execution of the `async` block until `futureDaysLeft` is completed.

The principle methods, `async` and `await`, have the following type signatures:

```
def async[T](body: => T): Future[T]
def await[T](future: Future[T]): T
```

Notably, `async` and `await` “cancel each other out:” `await(async { <expr> }) = <expr>`

**Reactive Extensions.** The Rx programming model is based on two interface traits: `Observable` and `Observer`. `Observable` represents observable streams, *i.e.*, streams that produce a sequence of events. These events can be observed by registering an `Observer` with the `Observable`. The `Observer` provides methods which are invoked for each kind of event produced by the `Observable`. In Scala, the two traits can be defined as follows:

```
trait Observable[T] { def subscribe(obs: Observer[T]): Closable }
trait Observer[T] {
  def onNext(v: T): Unit
  def onFailure(t: Throwable): Unit
  def onDone(): Unit
}
```

The idea of the `Observer` is that it can respond to three different kinds of events, (1) the next regular event (`onNext`), (2) a failure (`onFailure`), and (3) the end of the observable stream (`onDone`). Thus, the two traits constitute a variation of the classic subject/observer pattern [2]. Note that `Observable`’s `subscribe` method returns a `Closable`; `Closable` has only a single `close` method which removes the subscription from the observable.

### 3 Direct-style asynchronous observables

The following example demonstrates our programming model:

```
val filter = async*[Int] {
  var next: Option[Int] = await(input)
  while (next.nonEmpty) {
    val evt = next.get
    if (p(evt)) yield(evt)
    next = await(input)
  }
}
```

Here, we create a simple filter observable which publishes an `Int` event for each event observed on the `input` observable that satisfies predicate `p`.

We provide a complete formalization in the context of an object-based core language reminiscent of Creol [7] and ABS [6]. Figure 1 shows a subset of expressions of the core language.

$t ::=$	terms
...	(omitted)
<code>yield</code> ( $x$ )	yield event
$e ::=$	expressions
...	(omitted)
<code>async</code> *[ $\sigma$ ]( $\bar{y}$ ) { $e$ }	observable creation (reactive async)
<code>await</code> ( $x$ )	await event
$t$	term

Figure 1: RAY expressions and terms.

**Operational semantics.** The core concepts of our operational semantics are heaps, frames, and frame stacks (threads). Frames have the form  $\langle L, e \rangle^l$  where  $L$  maps local variables to their values,  $e$  is an expression, and  $l$  is a label. A label is either  $s$  denoting a regular, synchronous frame, or  $a(o, \bar{p})$  denoting an asynchronous frame; in this case,  $o$  is the heap address of a corresponding observable object, and  $\bar{p}$  is a sequence of object identifiers of observables that observable  $o$  has itself subscribed to.

**Correctness properties.** We show that well-typed programs satisfy desirable properties:

1. *Observable protocol.* For example, a terminated observable never publishes events again; this protocol property is captured by a *heap evolution* invariant which generalizes an invariant given in [1].
2. *Subject reduction.* Reduction of well-typed programs preserves types.

The proofs of these properties are based on a typing relation, as well as invariants preserved by reduction. A forthcoming technical report [3] provides details of the formal model and proofs.

## References

- [1] Gavin M. Bierman, Claudio V. Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. Pause ‘n’ play: Formalizing asynchronous C#. In *ECOOP*, volume 7313, pages 233–257. Springer, 2012.
- [2] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
- [3] Philipp Haller and Heather Miller. A formal model for direct-style asynchronous observables. Technical report, forthcoming, KTH Royal Institute of Technology, Sweden, 2015.
- [4] Philipp Haller and Jason Zaugg. SIP-22: Async. <http://docs.scala-lang.org/sips/pending/async.html>, 2013.
- [5] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde, editors. *The C# Programming Language*. Addison-Wesley, fourth edition, 2011.
- [6] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, volume 6957, pages 142–164. Springer, 2010.
- [7] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):39–58, 2007.
- [8] Erik Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, 2012.
- [9] Don Syme, Tomas Petricek, and Dmitry Lomov. The F# asynchronous programming model. In Ricardo Rocha and John Launchbury, editors, *PADL*, pages 175–189. Springer, 2011.

# Probabilistic Floating-Time Transition System: A New Approach For State Space Reduction of Probabilistic Actors

Ali Jafari<sup>1</sup>, Ehsan Khamespanah<sup>1,2</sup>, Marjan Sirjani<sup>1</sup>  
and Holger Hermanns<sup>3</sup>

<sup>1</sup> Reykjavik University, School of Computer Science and CRESS  
ali11@ru.is, ekhamespanah@yahoo.com, marjan@ru.is

<sup>2</sup> University of Tehran, School of ECE

<sup>3</sup> University of Saarland, School of Computer Science  
hermanns@cs.uni-saarland.de

Probabilistic Timed Rebeca language was proposed in [1] as an extension of Timed Rebeca language. PTRebeca is an actor-based modeling language that supports modeling of timing, probabilistic and non-deterministic features of real-time systems. In [1], the semantics of PTRebeca is presented as a timed Markov decision process (TMDP) which can be regarded as a discrete-time semantics of a probabilistic timed automaton (PTA). We also developed a supporting tool for formal analysis of PTRebeca models which uses the back-end model checker PRISM to provide performance evaluation of models.

The state space generated from TMDP representation of PTRebeca models suffer from the state space explosion problem. In our semantics, the execution of statements of message servers are interleaved from various actors concurrently being executed in the real-time system. The semantics also includes a discrete global time and probabilistic information which make the state space explosion problem even more serious. In the semantics, the local time of all actors progresses in a lock step manner with the global time.

In [2], authors proposed floating time transition system (FTTS) as a solution of the state space explosion problem in model checking of Timed Rebeca models. In FTTS, actors proceed with their own rates with independent local clocks instead of synchronizing with the global time. In Timed Rebeca language, and consequently in PTRebeca language, actors can request a service from other actors by sending a message to them; each actor has a bag of messages which stores the received messages. The receiver actor takes a message from its bag and executes its corresponding message server to provide the requested service. In FTTS, by taking a transition, all statements of a message server of an actor are executed and the execution result is available in the next state. The execution of statements of the message server do not interleave with the execution of statements of other message servers from other actors. Since the message server may include timed statements, the local time of actors can have different values in a state. Relaxing the synchronization of progress of time among actors and the complete execution of a message server in a step avoid many interleaves and result in a significant state space reduction in FTTS.

In this work, we propose a probabilistic version of FTTS, called PFTTS, as a new semantics for PTRebeca language. Similar to FTTS, the proposed semantics reduces the state space significantly in comparison to TMDP semantics. Our intuitive understanding is that for a given PTRebeca model, its TMDP (aka PTTS) interpretation and its PFTTS interpretation are probabilistic trace-distribution equivalence, but we do not have a formal proof yet. Therefore, there is no LTL-without-Next formula which distinguishes two semantics in the sense that the min/max probabilities are the same for whatever formula is picked. As each action is not

Problem	Size	Using PFTTS			Using TMDP			Reduction	
		#states	#trans	time	#states	#trans	time	#states	#trans
Ticket Service	<b>1 customer</b>	13	17	< 1 sec	19	23	< 1 sec	32%	27%
	<b>2 customers</b>	315	417	< 1 sec	515	625	< 1 sec	39%	34%
	<b>3 customers</b>	3694	4949	2 sec	5962	7462	3 sec	39%	34%
	<b>4 customers</b>	23799	33617	11 sec	39528	51372	22 sec	40%	35%
	<b>5 customers</b>	92431	137041	72 sec	156919	212211	144 sec	41%	35%
Sensor Network	<b>1 sensor</b>	280	542	< 1 sec	343	605	< 1 sec	19%	19%
	<b>2 sensor</b>	3426	7119	1 sec	4254	7947	1 sec	20%	19%
	<b>3 sensor</b>	33959	79007	11 sec	40321	85369	15 sec	16%	8%
	<b>4 sensor</b>	211579	568603	83 sec	241011	598035	114 sec	12%	5%
TinyOS	<b>1 sensor</b>	2302	3100	< 1 sec	4507	5443	1 sec	49%	43%
	<b>2 sensor</b>	22630	38101	5 sec	48155	66765	13 sec	53%	43%

Table 1: Number of states and transitions, time consumption, and reduction ratio in model checking based on PFTTS and TMDP.

logged in the traces of TMDP, internal actions are not logged in the traces, LTL-without-Next properties are preserved. Therefore, model checking algorithms proposed for LTL properties can be applied to PFTTS instead of TMDP.

We also examined different case studies in different sizes to show that PFTTS generates fewer states and transitions in comparison to TMDP semantics. Apparently, the needed time for state space generation is decreased. To this goal, a toolset was developed to generate the state space of PFTTS and TMDP for a given PTRebeca model. The PTRebeca models of Ticket Service, Sensor Network, and TinyOS examples are accessible from [3]. Table 1 shows the experimental results.

Here, we briefly explain the first case study. There are a customer, a ticket server, and an agent. The customer sends a ticket request by sending a message to the agent. The agent forwards the request to the ticket server. The ticket server issues a ticket and replies to the agent request, and then the agent sends the ticket to the customer. The customer sends a new request after a specified amount of time. In order to have different sizes of the ticket service model, there are different number of customers in the model varying from one to five customers. We aim at investigating how much PFTTS can reduce the state space size. The experimental results show that the state space enlarges quickly in accordance with the number of rebecs (i.e. actors like customer, agent, and ticket server) in the model.

## References

- [1] Ali Jafari, Ehsan Khamespanah, Marjan Sirjani, and Holger Hermanns. Performance analysis of distributed and asynchronous systems using probabilistic timed actors. In *14th International Workshop on Automated Verification of Critical Systems (AVoCS)*, volume 70, 2014.
- [2] Ehsan Khamespanah, Zeynab Sabahi Kaviani, Marjan Sirjani, Ramtin Khosravi, and Mohammad-Javad Izadi. Timed Rebeca Schedulability and Deadlock Freedom Analysis Using Bounded Floating-Time Transition System. In *Journal of Science of Computer Programming*, 2014.
- [3] Rebeca. Rebeca homepage. <http://www.rebeca-lang.org>.

# Towards Small-step Compilation Schemas for SOS

Ferdinand Vesely

Department of Computer Science, Swansea University, Swansea SA2 8PP, UK  
 csfvesely@swansea.ac.uk

## Abstract

We present work in progress on a method of compiling programs based on SOS specifications. The idea is to compile programs using SOS rules by translation into labelled blocks with explicit exit points, which implement a valid computation in the LTS of the program. Under this approach, a correct compiler can be constructed in a systematic way, based on an SOS specification.

## 1 Introduction and Background

Small-step SOS is a popular framework for specifying semantics of programming and specification languages. A collection of SOS rules together define the transitions of a labelled transition system (LTS). For programming languages, the states of an LTS usually contain a program term along with auxiliary entities (stores, environments) and labels may contain emitted signals or output streams. Table 1 contains a small example specification. Under such a specification, each program is represented by a concrete LTS. We present a compilation method which can be understood as the translation of the LTS to a corresponding control-flow graph (CFG). The nodes of the CFG are sequences of instructions with behaviour that should be equivalent to the states in the LTS.

**Atomic Blocks** Our method produces a collection of labelled *atomic blocks* (AB) containing instructions for a virtual machine. Each AB corresponds to a state in the LTS of the program, and is essentially a *basic block*: a sequence of instructions with one entry (at the beginning) and one exit point (at the end) [1]. However, we relax the second condition and allow multiple exit points at the end of the block, while requiring that an AB executes atomically as a single unit.

**Target Machine Language** We are targeting a simple register machine, with an unlimited supply of temporaries (registers). In this regard it is similar to LLVM [3], which we intend to

---

$\frac{\rho \vdash s_1 \xrightarrow{l} s'_1}{\rho \vdash \mathbf{let}(i, s_1, s_2) \xrightarrow{l} \mathbf{let}(i, s'_1, s_2)} \quad (1)$	$\frac{\mathbf{Value } v_1 \quad \rho[i \mapsto v_1] \vdash s_2 \xrightarrow{l} s'_2}{\rho \vdash \mathbf{let}(i, v_1, s_2) \xrightarrow{l} \mathbf{let}(i, v_1, s'_2)} \quad (2)$
$\frac{\mathbf{Value } v_2}{\rho \vdash \mathbf{let}(i, v_1, v_2) \xrightarrow{\tau} v_2} \quad (3)$	$\frac{\rho(i) = v}{\rho \vdash \mathbf{bound}(i) \xrightarrow{\tau} v} \quad (4)$
$\frac{\rho \vdash s \xrightarrow{l} s'}{\rho \vdash \mathbf{print}(s) \xrightarrow{l} \mathbf{print}(s')} \quad (5)$	$\frac{\mathbf{Value } v}{\rho \vdash \mathbf{print}(v) \xrightarrow{\mathbf{out } v} \mathbf{skip}} \quad (6)$

---

Table 1: Example language specification. ‘**Value**  $s$ ’ asserts that ‘ $\langle \rho, s \rangle$ ’ is a value (terminal) state for any  $\rho$ . As usual,  $\rho \vdash s \xrightarrow{l} s'$  is a shorthand for  $\langle \rho, s \rangle \xrightarrow{l} \langle \rho, s' \rangle$ .

use as the ultimate target. There is no program counter, instead the program is stored as a set of labelled code blocks  $\beta$ . Each block is (just) a sequence of instructions ‘ $\iota_1 \cdot \iota_2 \cdot \dots$ ’. The basic control-flow instructions are **halt** for stopping the machine immediately, and **jump** for unconditionally jumping to a labelled AB.

$$\frac{}{\beta \vdash \mathbf{halt} \cdot t \xrightarrow{\tau} \mathbf{halt}} \quad \frac{\langle l, t_2 \rangle \in \beta}{\beta \vdash \mathbf{jump} \ l \cdot t_1 \xrightarrow{\tau} t_2}$$

Further instructions will be mentioned in our translation example in the next section.

## 2 A Small-step Compilation Schema

Let’s take a simple construct like **print**. If there is a sequence of  $n$  transitions starting from term  $t$ , then the computation starting from **print**( $t$ ) will look as follows:

$$\begin{array}{ccccccccccc} t & \xrightarrow{L_1} & t_1 & \xrightarrow{L_2} & \dots & \xrightarrow{L_{n-1}} & t_{n-1} & \xrightarrow{L_n} & v & & \\ \mathbf{print}(t) & \xrightarrow{L_1} & \mathbf{print}(t_1) & \xrightarrow{L_2} & \dots & \xrightarrow{L_{n-1}} & \mathbf{print}(t_{n-1}) & \xrightarrow{L_n} & \mathbf{print}(v) & \xrightarrow{\{\mathbf{out}=v, \dots\}} & \mathbf{skip} \end{array}$$

The ABs for **print**( $t$ ) should each corresponds to a term (state) in the lower part of the above sequence. We construct a *translator* which will generate code blocks that implement the steps of the construct. A translator for construct  $f$ ,  $\mathbf{tr}_f$ , is a structure of operations **next**, **code**, and **label**. A *translator state* is constructed by applying  $\mathbf{tr}_f$  to translator states for arguments of  $f$ . We also write  $\llbracket f(t_1, \dots, t_n) \rrbracket$  for  $\mathbf{tr}_f(\llbracket t_1 \rrbracket, \dots, \llbracket t_n \rrbracket)$ , where  $n$  is the arity of  $f$ . For a translator  $tr$ , **next**  $tr$  is the next translator state, **code**  $tr$  is a code block corresponding to the current state, and **label**  $tr$  assigns a name to the state. The name can be used as a label for the atomic block or as the name of a temporary holding the computed value. The value of **next**  $tr$  can be **none** if the current state is final. In that case, the instructions in **code**  $tr$  must store a value in the named temporary **label**  $tr$ . The translator for **print**,  $\mathbf{tr}_{\mathbf{print}}$ , can be defined as follows:

$$\mathbf{code} \llbracket \mathbf{print}(t) \rrbracket = \begin{cases} \mathbf{code} \llbracket t \rrbracket & \text{if } \mathbf{next} \llbracket t \rrbracket \neq \mathbf{none} \\ \mathbf{code} \llbracket t \rrbracket \cdot \mathbf{out} \ \mathit{temp} & \text{otherwise, } \mathit{temp} = \mathbf{label} \llbracket t \rrbracket \end{cases} \quad (7)$$

$$\mathbf{next} \llbracket \mathbf{print}(t) \rrbracket = \begin{cases} \mathbf{tr}_{\mathbf{print}}(\mathbf{next} \llbracket t \rrbracket) & \text{if } \mathbf{next} \llbracket t \rrbracket \neq \mathbf{none} \\ \llbracket \mathbf{skip} \rrbracket & \text{otherwise} \end{cases} \quad (8)$$

A translator for a value  $v$  just has to put (a representation of) the value to a temporary, for which we introduce an instruction **ldval**.

$$\mathbf{code} \llbracket v \rrbracket = \mathbf{ldval} \ \mathit{temp} \ v \quad \mathbf{next} \llbracket v \rrbracket = \mathbf{none} \quad \mathbf{label} \llbracket v \rrbracket = \mathit{temp} \quad (9)$$

(where  $\mathit{temp}$  is a fresh temporary name)

The role of a top-level translator  $\mathbf{tr}_{\mathbf{top}}$  is to take the code block for each step and turn it into an atomic block by appending an explicit exit point (**jump** or **halt**):

$$\mathbf{code} \llbracket t \rrbracket_{\mathbf{top}} = \begin{cases} \mathbf{code} \llbracket t \rrbracket \cdot \mathbf{jump} \ l & \text{if } \mathbf{next} \llbracket t \rrbracket \neq \mathbf{none} \text{ and } l = \mathbf{label}(\mathbf{next} \llbracket t \rrbracket) \\ \mathbf{code} \llbracket t \rrbracket \cdot \mathbf{halt} & \text{otherwise} \end{cases} \quad (10)$$

$$\mathbf{next} \llbracket t \rrbracket_{\mathbf{top}} = \mathbf{next} \llbracket t \rrbracket \quad \mathbf{label} \llbracket t \rrbracket_{\mathbf{top}} = \mathbf{label} \llbracket t \rrbracket \quad (11)$$



The main compilation function just collects all the atomic blocks for a term, and returns them together with the initial block label:

$$\mathcal{C}(s) = \{\langle \text{label}[[t]]_{\text{top}}, \text{fold}_{\text{tr}}[[t]]_{\text{top}} \rangle\} \quad (12)$$

where

$$\text{fold}_{\text{tr}} tr = \begin{cases} \{\langle \text{label } tr, \text{code } tr \rangle\} \cup \text{fold}_{\text{tr}}(\text{next } tr) & \text{if } tr \neq \text{none} \\ \emptyset & \text{otherwise} \end{cases} \quad (13)$$

As a further illustration, we look at a definition of `code` for **let**. The construct uses an updated context in the premise of the rule in Eq. (2). The resulting code block also has to provide a corresponding context for the sub-block. This context has to be explicitly constructed at the beginning of the premise transition and cleaned up at the end. In this definition we assume a value translator for sets of mappings ‘ $\{i \mapsto v\}$ ’ and machine operations for manipulating environments.

$$\text{code}[[\text{let}(i, t_1, t_2)]] = \begin{cases} \text{code}[[t_1]] & \text{if } \text{next}[[t_1]] \neq \text{none} \\ \text{code } tr_{iv} \cdot \text{push\_env } tmp \cdot \text{code}[[t_2]] \cdot \text{pop\_env} & \text{if } \text{next}[[t_1]] = \text{none}, \\ & \text{next}[[t_2]] \neq \text{none}, \\ & tr_{iv} = [\{i \mapsto t_2\}], \\ & tmp = \text{label}(tr_{iv}) \\ \text{code}[[t_2]] & \text{otherwise} \end{cases} \quad (14)$$

### 3 Conclusion

We have illustrated a schema for small-step compilation on a few simple programming constructs. For lack of space we didn’t illustrate translations for, e.g., conditional, iterative, or non-deterministic constructs. The approach could be used with a suitable notion of bisimulation: to prove its correctness, to develop a compiler calculation method (following [2]), and to explore (semi-) automatic compiler generation based on SOS rules. To this end, we intend to work with Modular SOS [4], a modular variant of SOS, which places all auxiliary entities into labels of transitions, and the corresponding notions of bisimulation [5]. To deal with inherent inefficiencies (e.g., construction and destruction of contexts in atomic blocks), common optimisation methods, such as peephole optimisation, could be applied to the resulting translations.

### References

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [2] P. Bahr and G. Hutton. Calculating Correct Compilers. To appear in *J. Fun. Program.*, 2015.
- [3] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. CGO ’04, Washington, DC, USA, 2004. IEEE Computer Society.
- [4] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.
- [5] P. D. Mosses and F. Vesely. Weak bisimulation as a congruence in MSOS. In N. Martí-Oliet, P. C. Ölveczky, and C. Talcott, editors, *Logic, Rewriting, and Concurrency*, volume 9200 of *LNCS*, pages 519–538. Springer, 2015.

# Acyclic attribute evaluation in a dependently typed setting

Denis Firsov and Tarmo Uustalu

Institute of Cybernetics at TUT, Tallinn, Estonia, {denis, tarmo}@cs.ioc.ee

## 1 Introduction

Lately there has been some interest in certified parsing of context-free grammars. Attribute grammars extend context-free grammar with a semantics of parse trees in a declarative way. Let  $G = \langle N, \Sigma, P, F \rangle$  be a context-free grammar. An attribute grammar extends  $G$  by specifying two disjoint sets for each nonterminal  $X \in N$ , namely  $I(X)$  is a set of inherited and  $S(X)$  a set of synthesized attributes. Let  $A(X) := I(X) \cup S(X)$ ; then, for each production  $X \rightarrow Y_1 \dots Y_n$ , every synthesized attribute in the set  $S(X)$  has its value defined in terms of  $A(Y_1) \cup \dots \cup A(Y_n) \cup I(X)$  by so-called semantic equations. Similarly, each inherited attribute  $I(Y_i)$  has its value defined in terms of  $A(X) \cup S(Y_1) \cup \dots \cup S(Y_n)$ .

For example, we could specify the semantics of binary strings by the attribute grammar in Figure 1. Each nonterminal in this grammar has a synthesized attribute  $v$  and an inherited attribute  $p$ . The value of the attribute  $v$  for nonterminal  $X$  represents the semantical value of the tree starting from  $X$ . The value of the attribute  $p$  represents the position of the lowest bit of the subtree in the global tree.

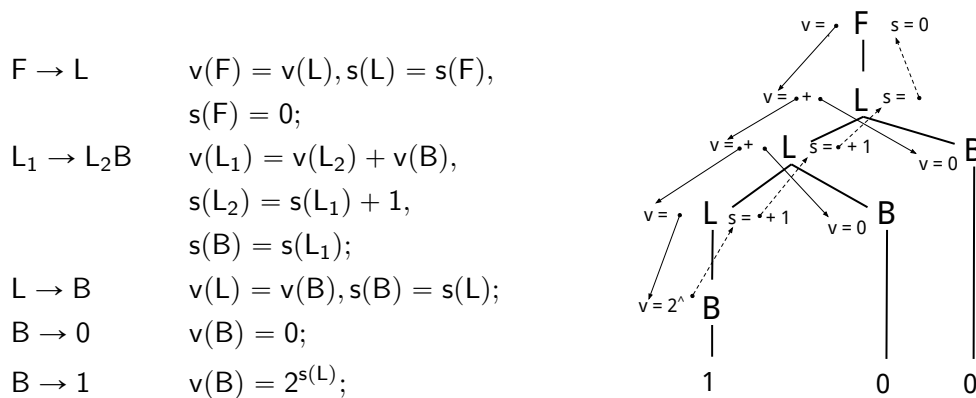


Figure 1: The attribute grammar specifies a parse tree traversal.

Figure 1 shows the parse tree of a string “100” and the evaluation of its semantic value. The semantics of that string is given by the value of attribute  $v$  of the start nonterminal  $F$  and equals 5. The value is computed by traversing the tree and applying the given semantic equations.

An attribute grammar is specification for attribute evaluation. However, it is easy to define grammars such that the dependency between attributes will be cyclic, causing lazy demand-driven attribute evaluation to diverge. An attribute grammar is called *cyclic*, if it is possible to construct a parse tree where an attribute of a particular node depends on itself.

Knuth [1] gave an algorithm for deciding whether an attribute grammar is cyclic or not. In this work we implement this algorithm in the Agda dependently typed programming language

together with proofs of correctness (soundness and completeness). This certified implementation of acyclicity checker is used to define a terminating evaluator for acyclic attribute grammars.

In this abstract, we focus our attention on designing the data structures and proving the principles that allow us to talk formally about cycles between attribute occurrences on parse trees.

## 2 Positions and paths in trees

In this section we will give some basic inductive definitions for trees, positions in trees and paths between these positions. In these definitions, we stay as general as possible and abstract away from specifics of attribute grammars such as production rules, attributes, attribute dependencies etc.

Our work is formalized in the constructive and dependently typed setting of the Agda language. But to avoid the notational clutter in definitions and theorems, we adopt an informal language, still relying on the Curry-Howard isomorphism and dependent type theory (propositions as types, proofs as programs).

**Definition 2.1** (Tree). *A finitely branching tree is given by a list of finitely branching subtrees (this definition is to be read inductively; the base case occurs when the list is empty).*

Next we define positions inside a tree. Each subtree of a tree occupies a certain position in the tree taken as a whole. The position consists of the subtree, along with the directions of how to navigate from the root to that location.

**Definition 2.2** (Position). *A position is a proof of a proposition  $\text{TreePos } t_1 \ t_2$  that states that  $t_2$  is a subtree of  $t_1$ . The inductive definition has two constructors:*

- *If  $t$  is a finitely branching tree, then  $\text{top } t$  is a position of type  $\text{TreePos } t \ t$ .*
- *If  $p$  is a position of type  $\text{TreePos } t \ t_1$  and  $c$  is a proof that a tree  $t_2$  is an immediate subtree of  $t_1$ , then  $\text{ins } p \ c$  is a position of type  $\text{TreePos } t \ t_2$ .*

**Definition 2.3** (Step). *A single step from a position  $p_1$  to a position  $p_2$  is a proof of a proposition  $[ p_1 ] \ d \ [ p_2 ]$  where  $d \in \{ \uparrow, \downarrow \}$ .*

- $[ p_1 ] \ \downarrow \ [ p_2 ]$  iff  $p_2 = \text{ins } p_1 \ c$  for some  $c$ .
- $[ p_1 ] \ \uparrow \ [ p_2 ]$  iff  $p_1 = \text{ins } p_2 \ c$  for some  $c$ .

Next, we would like to concatenate single steps from one position to another by taking the reflexive-transitive closure of single steps. However, we also want to state that a path is bounded by some position in the tree (it is confined to the corresponding subtree) and have a flag telling whether the path is empty (zero steps) or not. Having the bound parameter will allow us to have a unique decomposition of a cycle into smaller cycles.

**Definition 2.4** (Path). *A path from a position  $p_1$  to a position  $p_2$  bounded by a position  $b$  is a proof of a proposition  $b \mid [ p_1 ] \ f \ [ p_2 ]$  where  $f \in \{ \rightsquigarrow, \curvearrowright \}$ .*

- *An empty cycle  $p_1 \mid [ p_1 ] \ \curvearrowright \ [ p_1 ]$  is constructed by empty  $p_1$ .*
- *If  $s$  is a single step of type  $[ p_1 ] \ d \ [ p_2 ]$ ,  $p_2$  and  $p_1$  are bounded by some  $b$ , then  $\text{sngl } s$  is a path of type  $b \mid [ p_1 ] \ \rightsquigarrow \ [ p_2 ]$ .*
- *If  $s$  is a step  $[ p_1 ] \ d \ [ p_2 ]$ , and  $p$  is a path of type  $b \mid [ p_2 ] \ \rightsquigarrow \ [ p_3 ]$  for some  $b$ , and  $p_1$  is bounded by  $b$ , then  $\text{step } s \ p$  is a path of type  $b \mid [ p_1 ] \ \rightsquigarrow \ [ p_3 ]$ .*

It is clear that any path on a tree can be represented by a value of type  $b \mid [ p_1 ] \ f \ [ p_2 ]$  for appropriately chosen  $b$ ,  $p_1$ ,  $p_2$  and  $f$ . Cycles correspond to special cases when  $p_1 = p_2$ .

### 3 Decomposition and induction principle for cycles

Inductive types come with eliminators. We know that any natural number is either zero or successor of a smaller natural number and this deconstruction process cannot go on infinitely. The fact that a natural number can be deconstructed like this is the basis for proofs by induction. What about cycles on trees? To prove some property by induction for all cycles  $p \mid [p] f [p]$ , we need to argue from how cycles decompose into smaller cycles.

**Theorem 3.1** (Decomposition). *If  $c$  is a cycle of type  $p \mid [p] f [p]$  for some tree position  $p$ , then either  $c$  is empty ( $f = \hookrightarrow$ ) or  $c \equiv \text{sngl } sd \# c_1 \# \text{sngl } su \# c_2$  ( $- \# -$  concatenates paths), where:*

- $sd$  is a single step down of type  $[p] \downarrow [ \text{ins } p \ c ]$  for some  $c$ ;
- $c_1$  is a cycle of type  $(\text{ins } p \ c) \mid [ \text{ins } p \ c ] f [ \text{ins } p \ c ]$ ; note that it is bounded by a position one level lower than the original  $c$ ;
- $su$  is a single step up of type  $[ \text{ins } p \ c ] \uparrow [ p ]$ ;
- $c_2$  is another cycle of type  $p \mid [ p ] f [ p ]$ .

Moreover, this decomposition is unique.

Now we could prove an induction principle for cycles on trees.

**Theorem 3.2** (Induction principle). *Let  $P$  be a property of paths of type  $p \mid [p] f [p]$ , where  $p$  is any position in any tree. Then to conclude that  $P$  holds for all cycles on all trees, we need to establish the following:*

- $P$  holds for empty cycles on all trees.
- Any proofs that  $P$  holds for a cycle  $c_1$  of type  $(\text{ins } p \ c) \mid [ \text{ins } p \ c ] f [ \text{ins } p \ c ]$  and  $P$  holds for a cycle  $c_2$  of type  $p \mid [ p ] f [ p ]$  can be converted into a proof that  $P$  holds for the cycle  $\text{sngl } sd \# c_1 \# \text{sngl } su \# c_2$  where  $sd$  and  $su$  are steps down from and up to  $p$ .

## 4 Conclusion

As we already discussed, an attribute grammar specifies attribute evaluation on parse trees. We want to know if, for a given attribute grammar, it is possible to construct a parse tree with a cyclic attribute dependency. In this abstract, we described how cycles on trees can be decomposed into smaller cycles. Then we established that this decomposition implies an induction principle for cycles. The main components of our formalization are an acyclicity checker and an attribute evaluator. By using the decomposition theorem and induction principle, we prove that the acyclicity checker is correct (sound and complete). Our attribute evaluator works on attribute grammars coming with an acyclicity proof. Attribute evaluation is defined by a recursion that is wellfounded by acyclicity (the length of any acyclic path on a parse tree is bounded by the total number of attribute occurrences in this tree).

**Acknowledgement** This research was supported by the ERDF funded Estonian ICT national programme project “Coinduction”, the Estonian Science Foundation grant No. 9475 and the Estonian Ministry of Education and Research institutional research grant No. IUT33-13.

## References

- [1] D. E. Knuth. Semantics of context-free languages. In *Mathematical Systems Theory*, v. 2, pp. 127–148, 1968.

# Implementation-based Refinements for Equivalence Class Testing

Masoumeh Taromirad, Mohammad Reza Mousavi

Centre for Research on Embedded Systems (CERES),  
Halmstad University, Sweden  
[m.taromirad,m.r.mousavi]@hh.se

It is extremely laborious and ineffective to manually test complex software. Hence, test automation (automated test case generation and execution) has received significant attention. To be able to automatically generate test cases, a model (specification) of the system has to be available. Test cases are then automatically derived from this (preferably formal) model, and are executed on the system under test (SUT). This approach is typically known as model-based testing (MBT) [5].

The key artifact in MBT is a (formal) model that is used to describe a specification of the system. In an MBT process: (1) a model of the SUT is defined, (2) test selection criteria are specified, (3) test cases are automatically generated and executed, and (4) the final verdict of testing is produced.

A substantial challenge associated with MBT is that it is a black-box testing technique, in which the implementation is only accessible through its interfaces. Therefore, generated test suites may leave some untested gaps in a given SUT.

This is also a concern in test data selection. A common practice in generating specification-based test cases is to partition the input domain and then select test data from partitions, which are assumed to contain equally useful values from the testing perspective [1]. The theoretical foundations for dealing with the reduction of test suites have been laid out in [2] as the *regularity-* and the *uniformity hypothesis*. There are different testing techniques developed for partitioning and selecting representatives from the large input domain of parameters, such as equivalence-class based testing [3] and category-partition method [4]. In all of these techniques, partitions are derived from the reference model.

A promising approach to address this issue is to enrich initial test models with structural information exploited from the implementation domain, and then effectively generate concrete test cases and choose test data. With such test suites the coverage of the specification model and the implementation model would be complemented to each other.

We propose an approach to generate test cases considering both specification models and implementation models. The proposal is based on the input equivalence class partition (IECP) testing strategy presented by Huang and Peleska in [3]. We show that, under certain conditions, implementation models can be used in the equivalence class partition testing.

Our approach goes beyond the existing approaches by refining the initial in-out partitions obtained the specification using the structural information from the implementation. The advantages of this approach are twofold: firstly, we

cover the two models in tandem and hence avoid gaps in the specification or implementation. Secondly, we detect their possible structural differences (at interface level) during test case generation and steer the test case to exercise such possibly deviating paths of behavior.

## References

1. Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
2. Marie-Claude Gaudel. Testing can be formal, too. *TAPSOFIT. Lecture Notes in Computer Science*, 1995.
3. Wen ling Huang and Jan Peleska. Complete model-based equivalence class testing. *International Journal on Software Tools for Technology Transfer*, 2014.
4. Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 1988.
5. Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. *Journal Software Testing, Verification & Reliability*, 2012.









**REYKJAVÍK UNIVERSITY**  
HÁSKÓLINN Í REYKJAVÍK

---

School of Science and Engineering  
Reykjavík University  
Kringlan 1, IS-103 Reykjavík, Iceland  
Tel: +354 599 6200  
Fax: +354 599 6301  
<http://www.ru.is>  
ISSN 1670-5777