Workshop on Trustworthy Software

May 18–19, 2006, Saarbrücken, Germany

Edited by
Serge Autexier
Stephan Merz
Leon van der Torre
Reinhard Wilhelm
Pierre Wolper

Editors

Serge Autexier (DFKI Saarbrücken), autexier@dfki.de
Stephan Merz (INRIA Nancy & LORIA), Stephan.Merz@loria.fr
Leon van der Torre (University of Luxembourg), leon.vandertorre@uni.lu
Reinhard Wilhelm (Saarland University), wilhelm@cs.uni-saarland.de
Pierre Wolper (Université de Liege), pw@montefiore.ulg.ac.be

ACM Classification 1998
D.2.4 Software/Program Verification

ISBN 978-3-939897-02-6

Published online and open access by
Schloss Dagstuhl – Leibniz-Center for Informatics GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.

Publication date
August, 2006.

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/OASIcs.TrustworthySW.2006.i

ISBN 978-3-939897-02-6 **ISSN 2190-6807** **http://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 2190-6807

www.dagstuhl.de/oasics

Preface

The Interreg III C/E-Bird project "Recherches sans frontières/Forschen ohne Grenzen" aims at developing and strengthening the links between researchers in the SaarLorLuxWallonie region. Supporting actions initiated by the project were the documentation of shared and complementary research competences in that region as well as a series of thematic workshops held in 2006. The workshops were especially devoted to provide a forum for young scientists to present their research to a transnational audience from the SaarLorLuxWallonie region and to identify possible synergies and possibilities for cooperations.

The workshop on "Trustworthy Software" was the first workshop in that series and was held in Saarbrücken on 18-19th May 2006. It was organized by the Saarland University. As workshop theme the workshop chairs selected "Trustworthy Software" because a considerable concentration of research competence was found to exist in the SaarLorLuxWallonie region. The workshop aimed at presenting and fostering this research competence in the area of developing safe, secure and reliable software, computers and networks.

34 high-quality proposals were submitted for talks. In order to match the two-day format, the workshop chairs selected 21 for presentation at the workshop preferring contributions from young researchers.

The workshop consisted of the selected talks distributed into six sessions (one about Specification, three about Verification, one about Security and one about Privacy, Secrecy & Trust) and an invited talk by Christoph Weidenbach about "*From (Security) Protocol to Enterprise Network Infrastructure (Security) Analysis*". The detailed program is provided on the next page. These workshop proceedings incorporate a full paper or a short abstract for each talk.

We would like to thank several people who helped us in the organization of this workshop. First of all, many thanks to Wolfgang Lorenz and Signe Schelske, the coordinators for the project "Recherches sans frontières/Forschen ohne Grenzen", for their organizational and financial support. Many thanks also to Uta Merkle and their team for setting up the workshop environment and ensuring it to run smoothly. Last but not least, many thanks to all authors who submitted talks and to all active participants at the workshop.

Serge Autexier
Stephan Merz
Leon van der Torre
Reinhard Wilhelm
Pierre Wolper

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/693

Program

Session 1: Specification

Ina Schaefer	*Semantic-Based Modeling of Embedded Adaptive System*
Arnaud Lanoix	*An Operator-based Approach to Incremental Development of Conform UML 2.0 Protocol State Machines*
Axel Legay	*On the Implementation of a Game-based Model for Specifying Open Systems*
Julien Schmaltz	*Formalizing On Chip Communications in a Functional Style*

Session 2: Verification I

Joerg Bauer	*Analysis of Dynamic Communicating Systems by Hierarchical Abstraction*
Sebastien Varrette	*Applicative Solutions for Safe Computations in Distributed Environments*
Klaus Dräger	*Generation of linear synchronization invariants*

Session 3: Verification II

Antoine Reilles	*Formal Validation of Pattern Matching Code*
Jan Schwinghammer	*Separation Logic for General Storage*
Jan Reineke	*Shape Analysis of Sets*
Björn Wachter	*Explaining Data Type Reduction in the Shape Analysis Framework*

Session 4: Invited Talk

Christoph Weidenbach	*From (Security) Protocol to Enterprise Network Infrastructure (Security) Analysis*

Session 5: Verification III

Jan Dörrenbächer	*Formal Model and Verification of a Microkernel*
Thomas Hillenbrand	*Processor Datapath Verification with SPASS*
Jean-François Couchot	*Superposition Based Verification of Invariants. Application to Parameterized Systems.*
Artem Starostin	*Formally Verified Data Structures Library for C. The String Data Structure.*

Session 6: Security

Stephan Neuhaus	*Isolating Intrusions by Automatic Experiments*
Michael Hilker	*Security Analysis in Internet Traffic through Artificial Immune Systems*
Stefan Mandel	*Heuristics-based Source Code Analysis for Security Vulnerabilities*

Session 7: Privacy, Secrecy & Trust

J. Paul Gibson	*Trust and security in e-voting systems: the verification problem*
Eugen Zalinescu	*When reachability-based secrecy implies equivalence-based secrecy in security protocols*
Mathieu Turuani	*The CL-Atse Protocol Analyser*

Abstracts Collection
Workshop Trustworthy Software 2006
INTERREG IIIC/e-Bird

Serge Autexier, Stephan Merz, Leon van der Torre,
Reinhard Wilhelm and Pierre Wolper

Abstract. On 18-19 May 2006, the Saarland University organized a two-day workshop about "Trustworthy Software" in order to present and foster the research competence in the SaarLorLuxWallonie region in the area of developing safe, secure and reliable software, computers and networks. As part of the Interreg III C E-Bird project "Recherches sans frontières/Forschen ohne Grenzen" it provided an excellent forum especially for young scientists to present and discuss recent results, new ideas and future research directions to a transnational audience from the SaarLorLuxWallonie region. The workshop consisted of 21 regular presentations and one invited talk. Abstracts of all presentations are collected in this paper, including links to extended abstracts or full papers. The first section directs to the preface of the proceedings.

Keywords. Software evolution, Modularity, Automated debugging, Dependability assurance, Failure analysis, Static program analysis, Infinite and Finite-state verification, Runtime verification, Theorem proving, Access control, Security analysis, Security protocols, E-Voting

Preface – Workshop Trusworthy Software 2006

Serge Autexier (DFKI - Saarbrücken, D)

As part of the Interreg III C/E-Bird project "Recherches sans frontières/Forschen ohne Grenzen" the Saarland University organized a two-day workshop about "Trustworthy Software" in order to present and foster the research competence in the SaarLorLuxWallonie region in the area of developing safe, secure and reliable software, computers and networks. The workshop especially provided a forum for young scientists to present their research to a transnational audience from the SaarLorLuxWallonie region and consisted of 21 regular presentations and one invited presentation.

Keywords: Trustworthy software, preface

Joint work of: Autexier, Serge; Merz, Stephan; van der Torre, Leon; Wilhelm, Reinhard; Wolper, Pierre

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2006/693

Superposition Based Verification of Invariants. Application to Parameterized Systems.

Jean-François Couchot (Univeristé de Franche-Comté, F)

The harvey theorem prover implements a decision procedure for ground first order equational formulae in the array theory. This work provides valuable insights into the applicability of such a prover for the verification of safety properties expressed by an invariant on parameterized systems.

The soundness of such parameterized programs has to be checked uniformly, i.e. once for all its sizes. Such programs can be verified by a deductive fix point calculus whose proof obligations are discharged into a prover that allows quantified formulae.

Initiated by Graf and Saïdi who discharged their evolution conditions into the PVS prover, many studies have concerned the systems based on linear arithmetic constraints after a convenient counting abstraction.

We suggest a more basic but unifying approach in which the parameter ranges over a finite set. We show that such a framework is adequate for industrial test cases and uniform distributed systems. The specifications are written with the set theoretical B machine notation and we exploit an existing weakest precondition calculus for this method. Then, we introduce an invariant strengthening calculus, obtained as the refinement of a trivial calculus. We provide different methods for translating the evolution condition of this calculus into some equational logics. Their main objective is to make harvey discharge them as fast as possible.

On an industrial scale example, we show that this approach is more efficient than the Atelier B deductive system. Theoretically, we prove the evolution condition decidability, where the framework is some uniform distributed among broadcast and rendez-vous synchronization.

Keywords: Superposition, Verification, Parameterized Systems

SANA - Security Analysis in Internet Traffic through Artificial Immune Systems

Michael Hilker (University of Luxembourg, L)

The Attacks done by Viruses, Worms, Hackers, etc. are a Network Security-Problem in many Organisations. Current Intrusion Detection Systems have significant Disadvantages, e.g. the need of plenty of Computational Power or the Local Installation. Therefore, we introduce a novel Framework for Network Security which is called SANA. SANA contains an artificial Immune System with artificial Cells which perform certain Tasks in order to to support existing systems to better secure the Network against Intrusions. The Advantages of SANA are that it is efficient, adaptive, autonomous, and massively-distributed. In this Article, we describe the Architecture of the artificial Immune System and the Functionality of the Components. We explain briefly the Implementation and discuss Results.

Keywords: Artificial Immune Systems, Network Security, Intrusion Detection, Artificial Cell Communication, Biological-Inspired Computing, Complex Adaptive Systems

Joint work of: Hilker, Michael; Schommer, Christoph

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/694

An Operator-based Approach to Incremental Development of Conform Protocol State Machines

Arnaud Lanoix (LORIA, F)

An incremental development framework which supports a conform construction of Protocol State Machines (PSMs) is presented. We capture design concepts and strategies of PSM construction by sequentially applying some development operators: each operator makes evolve the current PSM to another one. To ensure a conform construction, we introduce three conformance relations, inspired by the specification refinement and specification matchings supported by formal methods. Conformance relations preserve some global behavioral properties. Our purpose is illustrated by some development steps of the card service interface of an electronic purse: for each step, we introduce the idea of the development, we propose an operator and we give the new specification state obtained by the application of this operator and the property of this state relatively to the previous one in terms of conformance relation.

Keywords: Protocol state machine, incremental development, development operator, exact conformance, plugin conformance, partial conformance

Joint work of: Lanoix, Arnaud; Okalas Ossami, Dieu-donné; Souquières, Jeanine

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/695

An Introduction to the Tool Ticc

Axel Legay (University of Liège, B)

This paper is a tutorial introduction to the sociable interface model of [?] and its underlying tool Ticc [?]. The paper starts with a survey of the theory of interfaces and then introduces the sociable interface model that is a game-based model with rich communication primitives to facilitate the modeling of software and distributed systems. The model and its main features are then intensivelly discussed and illustrated using the tool Ticc.

Keywords: Open system, game, interface automata

Joint work of: Legay, Axel; de Alfaro, Luca; Faella, Marco

Isolating Intrusions by Automatic Experiments

Stephan Neuhaus (Universität des Saarlandes, D)

When dealing with malware infections, one of the first tasks is to find the processes that were involved in the attack. We introduce Malfor, a system that isolates those processes automatically. In contrast to other methods that help analyze attacks, Malfor works by experiments: first, we record the interaction of the system under attack; after the intrusion has been detected, we replay the recorded events in slightly different configurations to see which processes were relevant for the intrusion. This approach has three advantages over deductive approaches: first, the processes that are thus found have been experimentally shown to be relevant for the attack; second, the amount of evidence that must then be analyzed to find the attack vector is greatly reduced; and third, Malfor itself cannot make wrong deductions. In a first experiment, Malfor was able to extract the three processes responsible for an attack from 32 candidates in about six minutes.

Keywords: Intrusion Analysis, Malware, Experimentation

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2006/696

Formal Validation of Pattern Matching code

Antoine Reilles (CNRS & LORIA, F)

When addressing the formal validation of generated software, two main alternatives consist either to prove the correctness of compilers or to directly validate the generated code. Here, we focus on directly proving the correctness of compiled code issued from powerful pattern matching constructions typical of ML like languages or rewrite based languages such as ELAN, MAUDE or Tom.

In this context, our first contribution is to define a general framework for anchoring algebraic pattern-matching capabilities in existing languages like C, Java or ML. Then, using a just enough powerful intermediate language, we formalize the behavior of compiled code and define the correctness of compiled code with respect to pattern-matching behavior. This allows us to prove the equivalence of compiled code correctness with a generic first-order proposition whose proof could be achieved via a proof assistant or an automated theorem prover. We then extend these results to the multi-match situation characteristic of the ML like languages.

The whole approach has been implemented on top of the Tom compiler and used to validate the syntactic matching code of the Tom compiler itself.

Keywords: Correctness proofs, compilers, pattern matching, validation

Joint work of: Kirchner, Claude; Moreau, Pierre-Etienne; Reilles, Antoine

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/697

Shape Analysis of Sets

Jan Reineke (Universität des Saarlandes, D)

Shape Analysis is concerned with determining "shape invariants", i.e. structural properties of the heap, for programs that manipulate pointers and heap-allocated storage. Recently, very precise shape analysis algorithms have been developed that are able to prove the partial correctness of heap-manipulating programs. We explore the use of shape analysis to analyze abstract data types (ADTs). The ADT Set shall serve as an example, as it is widely used and can be found in most of the major data type libraries, like STL, the Java API, or LEDA. We formalize our notion of the ADT Set by algebraic specification. Two prototypical C set implementations are presented, one based on lists, the other on trees. We instantiate a parametric shape analysis framework to generate analyses that are able to prove the compliance of the two implementations to their specification.

Keywords: Shape analysis, adt, algebraic specification, invariants, verification, set implementations, imperative programs

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/698

Using Abstraction in Modular Verification of Synchronous Adaptive Systems

Ina Schaefer (TU Kaiserslautern, D)

Self-adaptive embedded systems autonomously adapt to changing environment conditions to improve their functionality and to increase their dependability by downgrading functionality in case of fail- ures. However, adaptation behaviour of embedded systems significantly complicates system design and poses new challenges for guaranteeing system correctness, in particular vital in the automotive domain. Formal verification as applied in safety-critical applications must therefore be able to address not only temporal and functional properties, but also dynamic adaptation according to external and internal stimuli.

In this paper, we introduce a formal semantic-based framework to model, specify and verify the functional and the adaptation behaviour of syn- chronous adaptive systems. The modelling separates functional and adap- tive behaviour to reduce the design complexity and to enable modular reasoning about both aspects independently as well as in combination.

By an example, we show how to use this framework in order to verify properties of synchronous adaptive systems. Modular reasoning in com- bination with abstraction mechanisms makes automatic model checking efficiently applicable.

Keywords: Dependable Embedded Systems, Self-Adaptation, Abstraction, Modular Verification

Joint work of: Schaefer, Ina; Poetzsch-Heffter, Arnd

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/699

Formalizing On Chip Communications in a Functional Style

Julien Schmaltz (Universität des Saarlandes, D)

This paper presents a formal model for representing *any* on-chip communication architecture.

This model is described mathematically by a function, named *GeNoC*. The correctness of *GeNoC* is expressed as a theorem, which states that messages emitted on the architecture reach their expected destination without modification of their content. The model identifies the key constituents common to *all* communication architectures and their essential properties, from which the proof of the *GeNoC* theorem is deduced. Each constituent is represented by a function which has no *explicit* definition but is constrained to satisfy the essential properties. Thus, the validation of a *particular* architecture is reduced to the proof that its concrete definition satisfies the essential properties. In practice, the model has been defined in the logic of the ACL2 theorem proving system.

We define a methodology that yields a systematic approach to the validation of communication architectures at a high level of abstraction. To validate our approach, we exhibit several architectures that constitute concrete instances of the generic model *GeNoC*. Some of these applications come from industrial designs, such as the AMBA AHB bus or the Octagon network from ST Microelectronics.

Keywords: SoC's, NoC's, communication architectures, formal methods, automated theorem proving

Joint work of: Schmaltz, Julien; Borrione, Dominique

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/700

Separation Logic for General Storage

Jan Schwinghammer (Universität des Saarlandes, D)

Separation Logic is a substructural logic that facilitates local reasoning for imperative programs, in the sense that only the reachable part of the store must be taken into account for the verification of a command. In past work, Separation Logic has been developed for heaps containing records of basic data types.

Languages like C and ML, however, are less constrained and permit also the use of code pointers and higher-order references, respectively. The corresponding heap model is commonly referred to as "general storage" (or "higher-order store") since heaps may contain commands.

In this talk I will report on recent joint work with Bernhard Reus, where we make Separation Logic and the benefits of local reasoning available to languages with general storage.

Keywords: Program verification, Separation Logic, higher-order store

Explaining Data Type Reduction in the Shape Analysis Framework

Björn Wachter (Universität des Saarlandes, D)

Automatic formal verification of systems composed of a large or even unbounded number of components is difficult as the state space of these systems is prohibitively large. Abstraction techniques automatically construct finite approximations of infinite-state systems from which safe information about the original system can be inferred. We study two abstraction techniques shape analysis, a technique from program analysis, and data type reduction, originating from model checking. Until recently we did not properly understand how shape analysis and data type reduction relate. In this talk, we shed light on this relation in a comprehensive way. This is a step towards a more unified view of abstraction employed in the static analysis and model checking community.

Keywords: Canonical abstraction, data type reduction, model checking, parameterized system, infinite-state

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2006/701

Full Paper:
http://rw4.cs.uni-sb.de/∼bwachter/thesis.pdf

Relating two standard notions of secrecy

Eugen Zalinescu (UHP & LORIA & INRIA Project CASSIS, F)

Two styles of definitions are usually considered to express that a security protocol preserves the confidentiality of a data s. Reachability-based secrecy means that s should never be disclosed while equivalence-based secrecy states that two executions of a protocol with distinct instances for s should be indistinguishable to an attacker. Although the second formulation ensures a higher level of security and is closer to cryptographic notions of secrecy, decidability results and automatic tools have mainly focused on the first definition so far.

This paper initiates a systematic investigation of situations where syntactic secrecy entails strong secrecy.

We show that in the passive case, reachability-based secrecy actually implies equivalence-based secrecy for signatures, symmetric and asymmetric encryption provided that the primitives are probabilistic. For active adversaries in the case of symmetric encryption, we provide sufficient (and rather tight) conditions on the protocol for this implication to hold.

Keywords: Verification, security protocols, secrecy, applied-pi calculus

Joint work of: Zalinescu, Eugen; Cortier, Véronique; Rusinowitch, Michaël

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/691

Full Paper:
http://www.inria.fr/rrrt/rr-5908.html

An Introduction to the Tool Ticc*

Luca de Alfaro[1], Marco Faella[2], and Axel Legay[3]

[1] Department of Computer Engineering, Universitity of California, Santa Cruz, USA
[2] Dipartimento di Scienze Fisiche, Università di Napoli "Federico II", Italy
[3] Department of Computer Science, University of Liège, Belgium

Abstract. This paper is a tutorial introduction to the sociable interface model of [12] and its underlying tool TICC [1]. The paper starts with a survey of the theory of interfaces and then introduces the sociable interface model that is a game-based model with rich communication primitives to facilitate the modeling of software and distributed systems. The model and its main features are then intensivelly discussed and illustrated using the tool TICC.

1 Introduction

The prevalent trend in software and system engineering is towards component-based design: systems are designed by combining small components into bigger ones. Components offer thus the unit in which complex design problems can be decomposed, allowing the reduction of a single complex design problem into smaller design problems, more manageable in complexity, that can be solved in parallel by design teams. Components also provide a unit of reuse, defining the boundaries in which functionality can be packaged, documented and reused.

Components are designed to work as parts of larger systems: they make assumptions on their environment, and they expect that these assumptions will be met in the actual environment. In other words, a component is typically an open system which has some free inputs provided by others components and which in turn provides inputs to other components. It is thus obvious that the effective reuse of software requires adequate documentation of the component's behavior and the conditions under which it can be used along with methods for checking that components are assembled in an appropriate way. Such a documentation is commonly referred to as the *interface* of the component.

There have been many works on the design and implementation of good interfaces for components. Most of those works focus on capturing the *data dimension* of interfaces ("What are the value constraints on data communicated between components?") [21]. We describe here interface theories [13–15] a formal notion of component interfaces that use games to represent the interaction between the behavior originating within a component, and the behavior originating from the component's environment. Such an interface model is able to capture dynamic aspects of component interaction which makes it similar to a type system: indeed, it could be termed a "behavioral" type system for component interaction. In previous works, interface theories have been introduced

* This research was supported in part by the NSF grants CCR-0234690 and CCR-0132780, by the ARP awards SC2005553 and SC20051123, and by a F.R.I.A Grant.

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/766

for various aspects of component interaction: [13, 8, 7, 15, 12] consider the *protocol* dimension of interfaces ("What are the temporal ordering constraints on communication events between components?"), [16] considers the timing dimension of interfaces ("what are the real-time constraints on communication events between components?"), and [5] deals with constraints on the resource usage of the component.

In this paper, we focus on the *sociable interfaces* model introduced in [12], and on the corresponding tool called TICC [1]. We present the underlying ideas of the model, and show how it can be used to capture the protocol dimension between components. All the concepts are intensively illustrated with TICC for which this paper constitutes an introduction.

Two tools for interface theories predate TICC. The asynchronous, action-based interface theories of [13] are implemented as part of the Ptolemy toolset [19]. The tool CHIC [6] implements synchronous, variable-based interface theories modeled after [14]. Our goal in developing TICC is to provide an asynchronous model where components have rich communication primitives that facilitate the concise, natural modeling of software and distributed systems. In TICC, components are modeled both via variables (to describe state) and actions (to describe synchronization); its communication primitives enable the modeling of complex communication schemes. The implementation of TICC relies on symbolic methods, yielding efficient algorithms for component and system analysis.

2 Interface Theories

Before going to the details of the sociable interfaces model, we first summarize and illustrate the basic features of Interface theories. The reader is referred to [17, 10, 18, 12] for more details.

Interface Specification and Well-formedness

An interface specifies how a component interacts with its environment. It describes the input assumptions that the component makes on the environment and the output guarantees it provides. Interfaces capture the I/O behavior of a component by an automaton whose syntax is similar to the I/O automata of [21]. In the context of software design, inputs are used to model procedures or methods that can be called, and the receiving end of communication channels, as well as the return locations from such a calls. Outputs are used to model procedure or method calls, message transmissions, the act of returning after a call or method terminates, and exceptions that arise during method execution. Unlike traditional models of open systems, among which I/O automata, that at every state must be receptive to every possible input event, in interfaces it is possible that inputs are illegal (cannot be accepted) at some states. Thus, an interface describes the behavior of a component only with respect to some environments. In this way, environment restrictions can be used to encode restrictions on the order of method calls, and on the types of return values and exceptions. This is how interfaces capture the protocol dimension of components. Another advantage of making explicit assumptions about the environment is that it gives rise to an optimistic compatibility test when interface

are composed: two interfaces are compatible if there exists at least one environment in which they can work together. Finally, from a practical point of view, the ability to forbid inputs removes the need to specify "what happens" when taking an undesirable input. Such a specification has been pointed to as one of the main drawbacks of input-enabled approaches. Since we can make input assumptions, we have to ensure that the interface is *well-formed*, i.e. that there exists at least one environment that satisfies its input assumptions.

Interfaces as Games

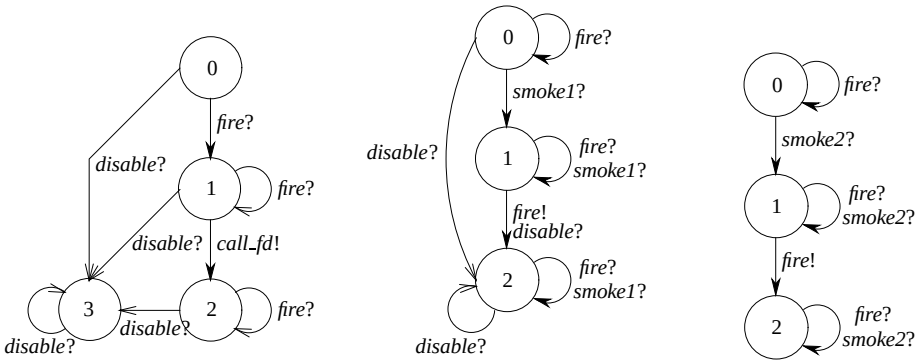
An interface is naturally modeled as a game between the players Input and Output. Input represents the environment: the moves of Input represent the inputs accepted from the environment. Output represents the component: the moves of Output represent the possible outputs generated by the component. Then, an interface is well-formed if the Input player has a winning strategy in the game, which means that the environment can meet all input assumptions. Games provides a model for multiple independent sources of nondeterminism and keep the distinction between inputs and outputs. Hence, even if the syntax of interfaces is close to the one of I/O automata, they differ in the way that the operations on the models are defined. In this paper, we will mainly mainly focus on the operation of composition between two or more interfaces.

Interface Composition and Compatibility

The game-like nature of interfaces becomes apparent when we consider the operation of composition. In their original formulation, interfaces interact through the synchronization of common input and output events. The interpretation of inputs and outputs as assumptions and guarantees, respectively, implies that, when composing two interfaces P and Q, we have to ensure that P's output guarantees satisfy Q's input assumptions and vice versa. Concretely, consider the two interfaces P and Q, in one state of the composition. If P wants to emit an output that cannot be accepted by Q in that state (i.e. an output guarantee that violates an input assumption), then a *local incompatibility* occurs. While many approaches would be pessimistic and consider the two interfaces to be incompatible, the interface approach is optimistic, by expecting the environment to steer away from locally incompatible states. Thus, two interfaces are *compatible* if there exists an environment to use the components together, and ensure that the assumptions of both are met. Component composition thus consists in synthesizing the most liberal input strategy in the composite system that avoids all locally incompatible states. This can be done by classical game-theoretic algorithms. The optimistic approach supports incremental design: the compatibility of two components can be checked without specifying interfaces for all components of the system, i.e. without closing the system. Incremental designs also ensure that compatible components can be put together in any order.

An Example

We illustrate the previous concepts with the help of a simple example: a fire detection system. The system is composed of a control unit and several smoke detectors. The



(a) C: Control Unit

(b) D_1: Smoke Detector

(c) D_2: Faulty Smoke Detector

Fig. 1. Sociable interface automata for a fi re detection system.

interfaces for this example are reported in Figure 1: D_1 is one of the smoke detectors (there could be more), and C is the control unit.

When a detector senses smoke (input event *smoke*?), it reports it by emitting the output event *fi re*!. When the control unit receives the input event *fi re*? from any of the detectors, it issues a call for the fire department (output event *call_fd*!). Additionally, an input event *disable*? disables both the control unit and the detectors, so that the smoke sensors can be tested without triggering an alarm. We also suppose the existence of faulty smoke detectors, i.e., smoke detectors that ignore the *disable* message. The interface for a faulty smoke detector is presented in Figure 1(c).

A particularity in the design is that some (but not all) states are input-enabled. As an example, state 1 of C is still receptive to the input event *fi re*? after receiving the smoke alarm. This is because detectors are independent and there is thus no reason for one detector to be forbidden to send output even *fi re*! if this has already been done by some other detector. Another example is state 2 of D_1 which is receptive to the input events *fi re*? and *smoke*$_1$?. Note that the possibility of having the same name for input and output events is proper of sociable interfaces model and not allowed in other interface models presented in [13, 8, 7, 15], or even transition based models (such as I/O automata). This illustrates the multiple ways of communicating that are allowed by the model (see [12] for a discussion). Note that C, D_1, and D_2 are well-formed[4].

It is easy to see that all the states in the composition between the control unit C and the fire detector D_1 will be compatible if the two interfaces communicate via the event *fi re*. As an example, if C is in state 1 and D_1 in state 2, then the output event *fi re*! emitted by D_1 can be caught by the input event *fi re*? of C. The output event *call_fd*! emitted by

[4] In general, checking well-formedness requires solving a safety game [12]

C does not need to be caught by D_1 since the two interfaces do not synchronize on this action.

However, the composition between C and D_2 goes less smoothly. When the composition receives the input event *disable*?, the control unit shuts down ($s = 3$) and makes the assumption that the environment cannot emit any output, while the faulty detector remains in operation. When the faulty detector senses smoke (input event *smoke*2?), it emits the output event *fi re*!: if the control unit has been disabled, this causes a local incompatibility in state $(3,1)$. Hence, a winning strategy for player Input to stay away from locally incompatible states can be realized by the following input restrictions:

- A restriction preventing the input event *disable*? if the faulty detector is in state $s = 1$, that is, it has detected smoke and is about to issue the output event *fi re*!.
- A restriction preventing the input event *smoke2*? when `ControlUnit` is at $s = 3$ (disabled).

Since the actions *disable* and *smoke2* should be acceptable at any time, the new input restrictions for these actions are a strong indication that the composition between C and D_2 does not work properly in all environments. However, if we consider an environment that never issues *disable*, then the two interfaces can work together in a proper way.

3 The Sociable Interfaces Model

This section sketches the main elements of a sociable interface and the game it induces. The reader is referred to [12] for more details. A sociable interface M is composed of the following elements.

- A set of *global actions* Act^G and a set of *local actions* Act^L.
- A set of *variables* V^{all} which is partitioned into a set of *local variables* V^L and a set of *global variables* V^G, with $V^G \cap V^L = \emptyset$. Local variables are used to describe the internal states of the interface, while global variables are used to describe the global state of the system. Among the set of global variables, we distinguish between *history* and *history-free* variables. This distinction, which will be discussed in Section 5, allows us to limit the number of actions an interface should include. The set of history variables is denoted by V^H.
- A set of input and output *transitions*. Each global action $a \in Act^G$ is associated to an input and an output transition that are respectively denoted by $\rho^I(a)$ and $\rho^O(a)$. An output transition specifies how variables are updated when the interface emits the action. An input transition is the conjunction of two parts: *(1)* an input global transition $\rho^{IG}(a)$ that specifies constraints on how other interfaces can update the global variables when emitting a, and *(2)* an input local transition $\rho^{IL}(a)$ that can update the local variables of the interface when other interfaces emit a. The reason to split the input transitions in two parts will be discussed in Sections 4 and 5.
- A set of *local transitions.* Each local action $a \in Act^L$ is associated with a transition $\rho^I(a)$, which can modify the value of local variables. Local transitions cannot be synchronized with transitions of other interfaces. Notice that local transitions were not present in the original model of [12].

– An input and an output invariant, respectively denoted by ψ^I and ψ^O. Invariants are sets of states that are used to constrain the input and output transitions of the interface. Precisely, input transitions must maintain the input invariant true, and output transitions must maintain the output invariant true.
– An initial condition I that describes the initial constraints on the set of local variables of the interface.

For an interface M, we say that a *state* of M is a value assignment to the variables in V^{all}.
Note that in TICC, the term "sociable interface" is replaced by "module", but a module is no more than the description of a sociable interface in the input language of the tool.

Example 1. The Control Unit C of the fire detection system described in the previous section is a sociable interface with 3 actions: *fire, disable, call_fd*. Its internal state can be encoded with a local variable s, so that a state of C assigns a value between 0 and 3 to s. The action *fire* has input transitions from states $s = 0, s = 1, s = 2$, but not from $s = 3$. The action has no output transition meaning that the interface cannot emit the action *fire*. The action *call_fd* has an output transition from $s = 1$ to $s = 2$, but no input transition. Hence, the interface makes the assumption that the environment can never issue *call_fd*. Notice that we can follow the same reasoning for the (faulty) smoke detectors.

The Game Underlying the Model

As mentioned in the introduction, a sociable interface induces a turn-based game between the Input and the Output players. The definitions of the moves, outcomes, and strategies of this game have been described in [12]. In this paper, the reader only needs to know how moves are defined.

The moves of the Input and Output players are those induced by the input and the output transitions (the game model supposes that both the input and the output transitions are conjoined with their corresponding invariants). In addition, each player owns a stuttering move to ensure that the runs of the game are infinite.

The definition of the stuttering move is straightforward for the Output player: this is the identity transition. For the Input player the definition is slightly different: the stuttering move is an additional transition that can modify the value of global history free variables. The stuttering move of the Input player is often referenced to as the *environment transition*; it is automatically added by TICC when specifying a sociable interface.

Well-Formedness

Given a sociable interface M, it is possible to compute the set of states S_I (resp. S_O) from which the Input (resp. Output) player has a strategy to always stay in the set of states that satisfy the input (resp. output) invariant, whatever the Output (resp. Input) player does. A sociable-interface is well-formed if each reachable state s of M belongs to $S_I \cap S_O$, and moreover $\psi^I = S_I$ and $\psi^O = S_O$; see [12] for a detailed explanation.

The tool TICC automatically ensures that modules are well-formed before allowing the user to manipulate them. To this end, TICC may add extra conditions to the initial condition and the input/output invariants that are defined by the user. Hence, the user does not need to take care of the notion of well-formedness, and we will not elaborate on it in the rest of the paper.

The Tool TICC

TICC is a tool that allows users to specify sociable interfaces, called "modules", using a textual language based on guarded commands, perform operations on the modules, and verify properties of modules. TICC is implemented as a set of functions that extend the capabilities of the OCaml [20] command-line. The tool is released under the GPL. The code of TICC is freely available and can be downloaded from *http://dvlab.cse.ucsc.edu/dvlab/Ticc*. This web site is a Wiki that also contains the documentation for the tool, as well as several examples including those that will be presented in this paper. Internally, TICC relies on a representation of modules which is based the MDD/BDD Glue and Cudd packages [22]. The source files of the tool are organized as follows:

1. The root contains files with the basic information about the tool. There is a `README` file that describes the files of the root.
2. The directory `examples` contains a series of examples and a tutorial. Again, there is a `README` file that can be consulted for more information.
3. The directory `src` contains the code itself; it is composed of three sub directories: `glu-2.0`, `mlglu`, and `ticc`. Directories `glu-2.0`, and `mlglu` contain the code needed to adapt the MDD/BDD Glu package to work with tcc. A directory `doc` contains automatically-generated documentation for the tool. In particular, the file `doc/api/Ticc.html` (automatically generated from `src/ticc/ticc.mli`) documents all the commands available to the user.

4 Starting with TICC

This section is an introduction to the use of TICC. It presents very simple examples, to illustrate the process of entering a program, and running the tool.

To use TICC, first ensure that the executable file "ticc" is in your path. Then, invoke it in interactive mode simply by typing:

```
ticc
```

The result of this operation is an Ocaml prompt[5] from where one must type:

```
open Ticc;;
```

[5] Remember that TICC is implemented as a set of functions that extend the capabilities of Ocaml.

At this point the functions in the module of TICC become available at the top level. These functions are documented in the file *ticc/doc/api/Ticc.html*. Most of them will be described in the rest of this paper.

The next operation is to provide TICC with a TICC program. TICC programs are entered in files with the extension .si that stands for **S**ociable **I**nterface. The syntax of TICC programs will be presented in the following sections. Program files are parsed with the command

```
parse "MyTiccProgram.si";;
```

The `parse` function reads in a .si file describing modules and global variables, and places these definition into a global namespace. If the .si file does not follow the syntax of the input language, the function reports an appropriate error message. Parsing multiple files is allowed and viewed as an incremental process: new declarations are added to the existing ones. This implies that one cannot declare two modules with the same name in different files, and that one only needs to declare global variables once. After parsing at least one TICC program, one can perform operations on and between modules of the program.

Notice that one can also write *script fi les* for TICC. A script file is a file that groups a set of commands that can be executed in one step. Figure 2(b) provides an example of the content of a script file whose name is `example.in`. At this point, the reader should be able to interpret lines 1 and 2. Lines 3 and 4 will be explained later. One can invoke TICC to execute the script file with the following command from the shell prompt:

```
ticc example.in
```

We dedicate the rest of this section to TICC programs that illustrate some of the main features of the input language of the tool. Operations on and between modules will be described in the next sections.

4.1 Getting to Know TICC Programs

As a first example, we consider the translation of the fire detection system to a TICC program. The file for the corresponding TICC program is given in Figure 2(a) and is named `detector.si`.

The program consists in the declaration of three modules: Module `ControlUnit`, `FireDetector1`, and `Faulty_FireDetector2` respectively correspond to interfaces C, D_1, and D_2 of Figure 1. Let us consider module `ControlUnit`. This module shows some of the very basic elements of a TICC module. It contains:

- Local variable declarations. The module declares a variable `s` whose value is an integer between 0 and 3. TICC supports Boolean and integer range variables.
- Input and output transitions. The transitions are specified using guarded commands *guard* $\Rightarrow$ *command*, where *guard* and *command* are Boolean expressions over the local and global variables; as usual, primed variables refer to the values after a transition is taken. For instance, the output transition `call_fd` can be taken only when s has value 1; the transition leads to a state where $s = 2$. The declaration of

```
module ControlUnit:
    var s: [0..3] // 0=waiting, 1=alarm raised, 2=fd called, 3=disabled

    input fire:   { local: s = 0 | s = 1 ==> s' := 1
                    else  s = 2       ==>       }
    input disable: { local: true ==> s' := 3 }
    output call_fd: { s = 1 ==> s' = 2 }
endmodule

module FireDetector1:
    var s: [0..2] // 0=idle, 1=smoke detected, 2=inactive

    input smoke1: { local: s = 0 | s = 1 ==> s' := 1
                   else  s = 2 ==>      } // do nothing if inactive
    output fire:  { s = 1      ==> s' = 2 }
    input fire:   { } // accepts (and ignores) fire inputs
    input disable: { local: true ==> s' := 2 }
endmodule

module Faulty_FireDetector2:
    var s: [0..2] // 0=idle, 1=smoke detected, 2=inactive

    input smoke2: { local: s = 0 | s = 1 ==> s' := 1
                   else  s = 2 ==>      } // do nothing if inactive
    output fire:  { s = 1      ==> s' = 2 }
    input fire:   { } // accepts (and ignores) fire inputs
    // does not listen to disable action
endmodule
```

(a) TICC modeling of a fi re detector system: `detector.si`.

```
open Ticc;;
parse "detector.si";;
let controlunit = mk_sym "ControlUnit";;
let faulty = mk_sym "Faulty_FireDetector2";;
```

(b) A script fi le that parses `detector.si`.

Fig. 2.

```
1  (* open the functionnalities of the tool *)
2  open Ticc;;
3
4  (* parse the file in where modules are described *)
5  parse "detector.si";;
6
7  (* create the symbolic representations for the three modules declared
       in fire-detector.si *)
8  let fire1 = mk_sym "FireDetector1";;
9  let faulty = mk_sym "Faulty_FireDetector2";;
10 let controlunit = mk_sym "ControlUnit";;
11
12 (* print the input and output invariants of symbolic module fire1 *)
13 print_symmod_iinv fire1;;
14 print_symmod_oinv fire1;;
15
16 (* print the transition rule corresponding to action "fire" in module
       fire1 *)
17 print_symmod_rules fire1 "fire" ;;
18
19 (* print the entire symbolic module fire1 *)
20 print_symmod fire1;;
```

Fig. 3. The TICC script `detector.in` for the fi re detector system.

the local part of an input starts with the keyword `local` (and so the global starts with `global`). This declaration has a particular structure, to ensure that the local part of the rule is deterministic (see next section for clarification).

The code of `detector.si` presents other features that will be extensively discussed in other examples.

An example of a script file for the fire detection system is given in Figure 3. The name of this file is `detector.in`. Let us briefly describe what happens when executing `ticc detector.in` from the shell.

Code between lines 1 and 5 has already been described earlier: we open the tool and parse a TICC program specified in a file called fire-detector.si. At this point, TICC contains an enumerative representation of the modules and the global variables that have been declared.

The command `mk_sym` used in lines 8, 9, and 10 converts the enumerative representation of modules into a symbolic representation based on MDDs [23]. An MDD is similar to a BDD [4], extended to work on integer ranged variables instead of Boolean ones. Given a constraint on a set of integer ranged variables, an MDD is a representation of all the values of the variables that satisfy the constraints.

The initial condition and the input/output invariants of a module are sets of constraints on its variables; they can thus be represented with MDDs. Since transition relations express constraints between the values of the variables before and after the

transitions have been applied, they can also be represented with MDDs. The symbolic representation is in general more compact and efficient than an enumerative one; TICC operations can be easily implemented symbolically, as explained in [12].

The rest of the file `detector.si` illustrates some of the printout functions available in TICC. As an example, in lines 13 and 14 the user asks TICC to print out the input and output invariants of the symbolic module `fire1`. In this example, both invariants have value `true`. In line 17 the user asks TICC to print the transition rule corresponding to action `fire` of module `FireDetector1`. The printout produced by this command is:

```
PRINTING the rule(s) for the action fire of
SYMBOLIC MODULE: FireDetector1.
[input part]:
modified vars:
{ }
[input global part]:
(1)
[input local part]:
(1)
[output part]:
Owned by module FireDetector1
modified vars:
{ FireDetector1.s }
(
 (FireDetector1.s = 1)(
   (FireDetector1.s' = 2)) )
```

When performing a printout, TICC describes the input and output transitions corresponding to the action, as well as the variables that are involved. Notice that a condition which is `true` is denoted by TICC as "`(1)`". For more printout functions, consult the documentation file *ticc/doc/api/Ticc.html*.

4.2 A More Elaborate Example

We now present a more elaborate example of TICC module, that makes use of most features of the input language. An Anti-blocking System (ABS) is an automotive component that tries to prevent wheel slippage by modulating the braking force. In Figure 4, we present a model of an abstract ABS, comprising two modules. Module `ABS_controller` is intended to be periodically invoked by the environment using the action `tick`. When it receives that action, the module moves to the internal state `state`=1 and sets the global variable `abs_on` to `true`. Then, it checks the current acceleration of the vehicle against the current pressure of the user on the brake pedal. If the module establishes that the situation requires ABS intervention, it emits action `do_it`, otherwise it goes back to internal state `state`=0 via the action `reset`.

Module `ABS_actuator`, instead, accepts an input signal `do_it`. At that time, it moves to a different internal state characterized by `state=true`. When `state=true`, the module controls the brakes according to a simplified anti-blocking algorithm.

In the following, let M be the sociable interface corresponding to module `ABS_actuator`.

```
1  var b_pedal, b_force: [0..5]
2  var accel: [0..10]
3  var abs_on: bool
4
5
6  module ABS_controller:
7     var state: [0..2]
8
9     stateless accel, b_pedal
10
11    initial: state = 0
12
13    input update_b_force: { global: abs_on ==> b_force' = b_force }
14    input tick: { global: abs_on ==> b_force' = b_force
15                  local: state = 0 ==> state' := 1
16                    else true ==> }
17    output do_it: {
18       state = 1 & (b_pedal > 0 & accel > 4) ==> state' = 2 & abs_on'
19    }
20    output reset: {
21       state = 1 & (b_pedal = 0 | accel <= 4) ==> state' = 0 & ~abs_on'
22    }
23    input done: { local: state = 2 ==> state' := 0 }
24
25 endmodule
26
27
28 module ABS_actuator:
29    var turn, state: bool
30
31    stateless b_pedal, b_force
32
33    initial: turn = false & state = false
34
35    oinv: true
36    iinv: true
37
38    input do_it: { local: ~state ==> state' := true }
39    output done: {
40       state & turn ==> b_force' = b_pedal & ~turn' & ~state';
41       state & ~turn ==> b_force' = 0    &  turn' & ~state'
42    }
43 endmodule
```

Fig. 4. TICC modeling of an Anti-blocking System.

Global variables. Global variables are declared outside modules. As we will see, multiple modules can read and modify the value of global variables.

In our case, the system comprises four global variables: `abs_on` indicates whether the ABS is currently controlling the brakes, `b_pedal` is the amount of pressure that the driver is currently applying on the brake pedal, `b_force` is the amount of pressure that the brake pads are currently applying to the brake rotors, and `accel` is the current acceleration of the vehicle. Since TICC does not support negative ranges, we assume that values of `accel` smaller than 4 represent negative accelerations.

In TICC, the set of global variables used by a module is automatically built by collecting all global variables that are mentioned in any transition rule. Thus, as far as module `ABS_actuator` is concerned, we obtain $V_M^G = \{\texttt{b\_pedal}, \texttt{b\_force}\}$.

History-free variables. By default, a module remembers the value of its global variables, and expects to know all actions that can modify them. More precisely, by default, global variables in a module are *history* variables. The module assumes that, unless some input or output action modifies their value, these global history variables retain their value through time. To enable reasoning about their value, if a global variable is a history variable in a module M, all the actions that can modify this variable must be known to M (declared as input).

This requirement can potentially require a module to possess very many input actions. There are two solutions to this problem. One, *wildcard actions*, will be described later. The other solution consists in declaring some variables to be *history-free*. In this case, the module does not track their value, and does not need to know (declare) all actions that modify their value.

In the case of module `ABS_actuator`, both `b_pedal` and `b_force` are declared to be history-free. `b_pedal` is naturally history-free, since we can make no assumptions on how the driver is going to use the brake pedal. `b_force` is also left history-free, as we assume that the actuator does not care if other modules change its value. Since no other global variable is mentioned by the module, we obtain $V_M^H = \emptyset$.

Local variables. Local variables are declared inside a module, using the same syntax of global ones. A local variable is only visible in the module it is declared in.

Module `ABS_actuator` declares two local variables of type bool, so that $V_M^L = \{\texttt{state}, \texttt{turn}\}$. `state` is true when the module is ready to emit its output action. `turn` is used to implement the following simplified anti-blocking algorithm: when `turn` is true, the actuator lets the driver decide the amount of braking, when `turn` is false, the actuator sets the braking force to zero.

Actions. In TICC, actions are not specifically declared. One can directly declare a transition rule and label it with a new or pre-existing action name. The tool collects all the actions used by a module in a set of module actions.
For module `ABS_actuator`, we have $Act_M^G = \{\texttt{do\_it}, \texttt{done}\}$ and $Act_M^L = \emptyset$.

Initial condition. A module can declare its initial condition using the keyword `initial`. The initial condition is expressed by a Boolean expression over the set of local variables.

In our case, module `ABS_actuator` starts with `turn` and `state` equals to `false`.

```
var x, y: [0..10]

module Test:
   oinv: x + y <= 15
   output a: { true ==> x' = x + 1 }
endmodule
```

Fig. 5. A module with a non-trivial output invariant.

Invariants. An invariant is a condition over the state space of a module, that is constantly satisfied. Following the input/output duality which is proper of interfaces, modules can have two invariants: an input invariant and an output invariant. The output invariant defines a set of states that will not be left by any local or output transition. In practice, each local or output transition rule is implicitly conjoined with the output invariant of the module. Dually, a module assumes that its environment does not violate its input invariant. In practice, all input transition rules are implicitly conjoined with the input invariant of the module. Note that, since modules are well-formed, the Input (resp. Output) player can ensure that the input (resp. output) invariant is never left. This indicates that no output transition leads from a state satisfying both invariants to a state satisfying the output, but not the input, invariant. Symmetrically, no input transition can lead from a state satisfying both invariants to a state satisfying the input, but not the output, invariant.

The invariants of `ABS_actuator` are both equals to `true`. In fact, specifying a `true` invariant is equivalent to specifying no invariant at all, as done by module `ABS_controller`. Invariants are useful to express certain relationships between variables. As instance, consider the example in Figure 5, comprising a module `Test`, together with two global variables.

The output invariant expresses the property that this module will always enforce that the sum of `x` and `y` is at most 15. This implies that module `Test` will not emit action `a` when the current sum of `x` and `y` is at least 15. As we will see later, the main use of invariants is in composition: input invariants will be used to express the constraints on the environment that guarantee the compatibility of the modules being composed.

Transition rules. TICC supports three types of transition: input, output and local transitions. Output transitions are the ones that users are most likely to be familiar with. They describe a possible behavior of the module, consisting in emitting an action, while possibly changing the value of global and local variables. Local transitions can be thought of as a special type of output transition, where the module is only allowed to update its local variables. Moreover, local transitions are invisible to other modules, so that the name of the action labeling a local transition is irrelevant. They can be declared using the syntax:

```
local a: { guard ==> command }
```

Module `ABS_actuator` can only emit one output action, called `done`. As previously said, the corresponding transition rule is expressed by a sequence of guarded

commands. In this case, the first guarded command (line 17) states that if both `state` and `turn` are true, action `done` can be performed. As a consequence, the next value of `b_force` will be equal to the current value of `b_pedal`, and both `state` and `turn` will have value false. The second guarded command (line 18) states that the transition can also be taken if state is true and turn is false. In this case, the next value of the global variable `b_force` will be zero, while the local variables `turn` and `state` will have value true and false, respectively. In this case, the two guards are mutually exclusive. In general, more than one guard can be true at a given time: at run-time, any of those guards can be selected nondeterministically.

Notice that action `done` occurs only as output in `ABS_actuator`. This implies that the module does not accept it as input.

One feature of TICC guarded commands that might surprise at first is that the distinction between guard and command is purely conventional. A guard and its corresponding command are internally conjoined, so that

```
guard ==> command
```

is always equivalent to:

```
true ==> guard & command
```

This holds for output rules, local rules, and the global section of input rules. The local section of input rules follows a different syntax, as explained later in this section.

For instance, consider again module `Test` in Figure 5. The transition rule corresponding to action "`a`" seems to state that module `Test` can always emit "`a`", whose effect will be to increase the value of `x`. However, according to the principle we just stated, the action cannot in fact be emitted when `x=10`.

Input transition rules are split in two sections. The *global* section describes assumptions about how other modules can change the value of global variables when emitting certain outputs. The *local* section describes how this module reacts when receiving a certain action. The reaction of the module to an input has two important restrictions: *(i)* it can only update local variables, and *(ii)* it must do so in a *deterministic* fashion. These restrictions are due to the theoretical assumption that each step is driven by the module carrying out the output action. In turn, this ensures that the semantics of the model is a turn-based game rather than a concurrent one. As a consequence, we have the following special syntax for the local part of input rules:

```
     guard1 ==> var11' := expr11, var12' := expr12, ...
else guard2 ==> var22' := expr21, var22' := expr22, ...
...
```

To ensure determinism, commands can only include assignments to local variables. Moreover, the `else` keyword is inserted to remind the user that in this context guarded commands will be evaluated in the order in which they are written, (i.e., `guard2` is evaluated only if `guard1` is false, and so on).

The only input action that module `ABS_actuator` can accept is called `do_it`. The corresponding transition rule has no global section, meaning that the module makes no assumptions on the current and next value of global variables when `do_it` is received. The local section states that, when `state` is false and `abs_on` is true, the next value

of `state` will be `true`. We may wonder what happens when the conditions set by the guard fail (i.e., `state` is true or `abs_on` is false). The answer is that the condition expressed by the guard becomes an input assumption and as such it *migrates* to the global part of the rule, as witnessed by a printout of the module. In other words, the input rule corresponding to action `do_it` is equivalent to the following:

```
input do_it: {
   global: ~state ==> true
   local: true ==> state' := true
}
```

4.3 Arithmetic in TICC

TICC allows the declaration of Boolean and integer range variables. Both of those declaration have previously been illustrated. However, due to the bounded size of the variables, dealing with integer range variables implies some implementation choices that are worth summarizing.

From the previous section, we learned that integer range variables allow to build numerical expressions, while Boolean variables allow to build Boolean expressions. The two types of expressions are combined in guarded commands with the classical Boolean and numerical comparison operators. The question arises of how to interpret the arithmetical operators $+$ and $-$ on a finite range type. A common choice is to implement modulo arithmetic: for instance,if x and y have range $[0..\ m-1]$, then the expression $x+y$ is evaluated to $x+y \mod m$. This is the choice followed, for instance, in Mocha [3, 11]. There are two drawbacks in following this choice. The first is that comparisons behave in a counterintuitive way, making the system prone to modeling errors. For instance, the two comparisons $x+1 \geq y$ and $x \geq y-1$ are *not* equivalent: the first returns an unexpected result with $x = 3$, the second when $y = 0$. The second drawback is that it is difficult to come up with consistent and intuitive typing rules for expressions including variables with different ranges; for instance, it is not clear how to evaluate $x+y+z = w$ if all of x, y, z, and w have different ranges. Indeed, the tool Mocha avoided this problem by forcing expressions to consist of one range type only, which is a rather restrictive requirement.

In TICC, we follow a different choice, based on the following two principles:

1. Numerical expressions are always evaluated in a range that is large enough so that no roll-over, or overflow, occurs.
2. Negative numbers are not considered.

Let us illustrate the consequences of these principles. Consider the expression:

```
x' = y + z - 3
```

and assume that the ranges are as follows:

```
var x: [0..4]
var y: [0..5]
var z: [0..5]
```

The design decisions imply that:

1. The sum of y and z is evaluated in a temporary range type that is at least $[0..10]$, so that no overflow can occur.
2. If the result of the expression is negative, it is considered different from the result of any other expression, and in particular x', so that the overall expression will be false.

The expression is thus evaluated as follows:

- If x is 4, y is 4, and z is 3, then the expression $x' = y + z - 3$ will be true, as expected. In fact, $4 + 3$ will give 7, and $7 - 3 = 4$: no overflow occurs.
- If x is 1, y is 4, and z is 5, the expression is false, as $4 + 5 - 3 = 6$, which is different from 1. Note in particular that roll-over does not occur: even though $6 \mod 5 = 1$, the expression on the right hand side is considered to have value 6, not 1, in spite of the left hand side having range $[0..4]$.
- If y is 1, and z is 1, the expression will be false, since the right hand side gives rise to a negative number.

The evaluation of an expression proceeds by evaluating sub-expressions and by combining the obtained results. In general, one could suppose that if a sub-expression is evaluated to false, then the entire expression is evaluated to false. As an example, consider the following expression:

```
x' = y - z + 3
```

If x' is 2, y is 2, and z is 3, then the expression would yield value false because $y - z$ represents a negative number. However, we have that $2 = 2 - 3 + 3$, meaning that the evaluation of the whole expression is true! To mitigate (but not eliminate) this, after parsing, TICC tries to reorder the expressions, so that whenever possible, negative results are avoided. For instance, the above expression would be internally transformed into the following expression:

```
x' = y + 3 - z
```

so that a negative result would occur only if the total result is negative. TICC can do basic expression simplification, and it reorders the terms of a sum so that positive terms occur before negative terms. A good way for the user to know if reordering occurred is to print the syntactic representation of a module after parsing it.

5 Composing Sociable Interfaces in TICC

In TICC, the main operation on modules is *composition*. Composition synchronizes two modules on their shared actions, and returns a new module, representing the joint behavior of the two original modules, along with the environment assumptions required to guarantee the correct functioning of the original modules. While composing modules, TICC checks their *composability* and *compatibility:*

```
open Ticc;;

parse "fire-detector-disable.si" ;;

let controlunit = mk_sym "ControlUnit";;
let fire1 = mk_sym "FireDetector1";;
let wfire2 = mk_sym "Faulty_FireDetector2";;

let c = compose fire1 controlunit;;
let d = compose wfire2 controlunit;;

print_symmod c;;
print_symmod d;;

print_input_restriction c "disable";;
print_input_restriction d "disable";;

print_input_restriction c "smoke1";;
print_input_restriction d "smoke2";;
```

Fig. 6. A script file illustrating the composition of the modules for the fire detector example of Figure 2(a).

- *Composability* is a condition involving the sets of variables and actions of a module, and that can be checked statically, and extremely efficiently. Essentially, two modules are composable if it makes sense to consider the effect of their communication.
- *Compatibility* is a condition about the behavior of the modules. Two modules are compatible if there is some environment in which they can work jointly together, with all their input assumptions being satisfied. Checking compatibility requires solving a game between the Input and Output player; the solution of the game yields the input assumptions for the composition of the two modules.

The TICC command `compose` checks composability and compatibility of two modules, and if both tests are positive, computes a symbolic module corresponding to their composition. If incompatibilities arise, TICC can provide diagnostic information to detect the reason.

Example 2. The script file given in Figure 6 illustrates the composition operation for the fire detector example mentioned in Section 2 and Figure 2(a).

In the sociable interface model, and thus in TICC, the composition is done in four steps. First, one checks that the modules can be composed (see Section 5.1). If the modules are composable, then the next step is to build the product between them (see Section 5.2). At this point, the product can contains bad states, i.e. states that exhibit a local incompatibility (see Section 5.3). The last step of the composition consists in

synthesizing a strategy for the Input player to stay away from the set of bad states whatever the Output player does (see Section 5.4).

This section describes how those four steps are conducted in TICC. More information about the theory behind the operations can be found in [12].

We remark that the composition of two modules in TICC only works on their symbolic representation. In what follows, we consider two symbolic modules M_1 and M_2 where $M_i = (Act_i^G, Act_i^L, V_i^G, V_i^L, V_i^H, \rho_i^I, \rho_i^O, \rho_i^L, \psi_i^I, \psi_i^O)$, and we implicitly refer to their corresponding sociable interfaces.

5.1 The Composability Condition

To facilitate composition, TICC ensures that modules have distinct local actions and local variables by automatically renaming local variables and local actions: a local variable x of module M is renamed to $M.x$ upon parsing the module M.

We say that the two modules M_1 and M_2 are *composable* if they satisfy the following non-interference condition: if an action $a \in Act_1^G$ (respectively Act_2^G) of module M_1 (resp. M_2) can modify a history variable of module M_2 (resp. M_1), then $a \in Act_2^G$ (resp. Act_1^G).

Since output transitions are the only ones that can modify the value of a global variable[6], the condition boils down to checking that if module M_1 has an output transition for action a that modifies[7] a history global variable of module M_2, then module M_2 must have an input transition for action a.

The non-interference condition is the main motivation for distinguishing between history and history-free variables. The non-interference condition states that a module should know all actions of other modules that modify its history variables. If we dropped the distinction, requiring that a module knows all actions of other modules that can change any of its variables (history or history-free), we could greatly increase the number of actions that must be known to the module. Wildcard actions, as described later, is another method.

Example 3. Consider the composition of the modules in the Anti-blocking System (ABS) described in Section 4.2. The global variable `b_force` is a history variable for module `ABS_controller`. Since module `ABS_actuator` has an output transition for action `done` that modifies this variable, module `ABS_controller` *must* accept `done` as input. In this case, the input transition of action `done` states that module `ABS_controller` agrees on all modifications that could be done to the variable.

Another consequence of the non-interference condition is the following. Denote *ABS* the module obtained by composing the two ABS modules. If another module wants to modify variable `b_force` and be composed with *ABS*, it is forced to do so using one of the remaining inputs of *ABS*, namely `tick` and `update_b_force`. Both those input transitions impose the condition that if `abs_on` is true, the value of `b_force` is not modified. Thus, the non-interference condition allows modules to effectively control a global variable, when needed.

[6] Input transitions only make assumptions on those values.

[7] Where "modifies" means that the the variable appears primed in the command of the output transition.

5.2 The Product

The product describes how elements of M_1 and M_2 are combined to give rise to a new module M_{12} representing their joint behavior.

First, the set of local, global, and history variables are obtained by taking the unions of those of the two modules: $V_{12}^{all} = V_1^{all} \cup V_2^{all}$, $V_{12}^{L} = V_1^{L} \cup V_2^{L}$, and $V_{12}^{H} = V_1^{H} \cup V_2^{H}$. The same stands for the set of actions: $Act_{12}^{G} = Act_1^{G} \cup Act_2^{G}$ and $Act_{12}^{L} = Act_1^{L} \cup Act_2^{L}$. The input and output invariants of M_{12} are obtained by conjoining those of M_1 and M_2, and so for the initial condition.

The most crucial part in the definition of the product concerns the transitions associated to the actions of M_{12}. Those transitions are a suitable combination of the transitions of M_1 and M_2.

Similarly to other interface models, for each shared action, the output transition of M_1 synchronizes with the input transition of M_2, and symmetrically, the output transition of M_2 is synchronized with the input transition of M_1. This models communication, and gives rise to output transitions in the product. The input transitions of M_1 and M_2 corresponding to the same shared action are also synchronized, and lead to an input transition in the product. Output transitions, on the other hand, are not synchronized between them: if both M_1 and M_2 can emit a shared action a, they do so asynchronously, so that their output transitions interleave. As usual, the modules interleave asynchronously on transitions labeled by non-shared actions. We now describe in more details the interleaving on shared actions.

If M_1 has an input transition $\rho_1^I(a)$, and M_2 has an input transition $\rho_2^I(a)$, then M_{12} has an input transition $\rho_{12}^I(a)$. The local and global part of $\rho_{12}^I(a)$ are obtained by conjoining those of ρ_1^I and ρ_2^I, i.e., $\rho_{12}^{IL}(a) = \rho_1^{IL}(a) \wedge \rho_2^{IL}(a)$ and $\rho_{12}^{IG}(a) = \rho_1^{IG}(a) \wedge \rho_2^{IG}(a)$. This models the fact that M_1 and M_2 can react jointly to inputs from the environment.

The situation is more complicated for output transitions. Suppose that M_1 has an output transition $\rho_1^O(a)$, and M_2 has an input transition $\rho_2^I(a)$. The result of the two transitions is an output transition $\rho_{12}^O(a)$ in M_{12}, obtained by conjoining $\rho_2^{IL}(a)$ with $\rho_1^O(a)$.

The reader could wonder why the new output transition is not obtained by conjoining also $\rho_2^{IG}(a)$ with $\rho_1^O(a)$. The reason is the definition of input and output transitions: output transitions can modify global variables, while input transitions can only make assumptions on them. The assumptions expressed by the global section of input rules will be taken into account in the next phase of composition.

5.3 Locally Incompatible States

The product defined in the previous section can contain *locally incompatible states.* In a locally incompatible state, one of the modules being composed wants to issue an output transition labeled by a shared action, while the other module does not have a corresponding global input transition from that state which agrees with the output transition on the updates of global variables. In practice, TICC computes the set of good states *Good*, which is simply the complement of the set of locally incompatible states.

Example 4. Consider the fire detector example of Section 2, illustrated in Figure 2(a). In the composition of `ControlUnit` and `Faulty_FireDetector2`, the state where

ControlUnit.s $= 3$ and Faulty_FireDetector2.s $= 1$ is locally incompatible: module Faulty_FireDetector2 can issue the output action fire, which module ControlUnit, being disabled, cannot accept.

5.4 Synthesizing a Strategy

After computing the product of the two modules and the set of good states, the next operations is to compute the set of states *Win* from which the Input player of M_{12} has a strategy to always stay in *Good*. This is done by playing a safety game whose objective is *Good*. The result of the game is used to restrict the input invariant of the product (use the command print_input_restriction to see how the new invariant restrict the Input transitions of the composition). Hence the composition of the two modules can only works in environments that satisfy the restricted input invariant. This can be considered an optimistic approach, since two modules are not considered to be incompatible if they cannot work in one particular environment.

The set *Win* is also conjoined with the initial condition of the product, giving rise to the initial condition of the composition. If the resulting initial condition is empty, the two modules are definitely incompatible.

Example 5. Consider again the fire detector example of Section 2, illustrated in Figure 2(a). The modules ControlUnit and Faulty_FireDetector2 are compatible: in fact, there is an environment that avoids all locally incompatible states. For instance, to avoid the state where ControlUnit.s $= 3$ and Faulty_FireDetector2.s $= 1$, the environment can simply avoid issuing the action smoke2 if disable has already been issued, or can avoid to issue action disable if smoke2 has already been issued.

Of course, such a compatibility masks the fact that it does not make sense to restrict the environment's ability to issue actions smoke2 — a fire can start at any time! The user can discover the problem by asking TICC to print the *restriction* of action smoke2, via the command

```
print_input_restriction d "smoke2";;
```

which generates the following output:

```
Restriction of input action smoke2:
(
 (Faulty_FireDetector2.s = 0)(
   (ControlUnit.s = 3)) )
```

This indicates that, after the composition, action smoke2 can no longer be accepted if no smoke has been detected yet (Faulty_FireDetector2.s = 0) and the controller has been disabled (ControlUnit.s = 3).

Similarly, the user can print the restriction of action disable in the composition of ControlUnit and Faulty_FireDetector2 to discover how the ability of accepting disable has been restricted by the composition.

6 Composition: A Concrete Example

In this section we present a concrete example of the use of TICC on a large program. We consider a model of the interaction among contractors fixing a house. The example illustrates how TICC can verify the compatibility of the interaction protocol among communicating entities.

The example models a house with four rooms: a K(itchen), a L(iving), a B(athroom), and a (Bed) R(oom). Each room can suffer from electrical and plumbing problems that can be fixed by a plumb(er) and an electr(ician). Depending of the problem that occurred, contractors are also needed to repair the damages caused on the wall and on the floor. After the repairs, the room has to be cleaned. As rooms are small, only one contractor at a time can work in a room.

We wish to know if the contractors can work together and fix the problems. This question can be answered in TICC by modeling each contractor as a module, and by considering additional modules that simulate faults, and that call the contractors to fix things. The contractors can work together if the composition of all the modules is compatible.

The TICC program corresponding to the example is as follows. Each room may have ongoing repair work; this is tracked by the following global variables:

```
var K_busy, L_busy, B_busy, R_busy: bool
```

In each room, four items might need repair: plumb(ing), electr(ical), floor, and wall. Moreover, the room may need to be clean(ed). For the kitchen, the need for repair and the need to clean are tracked by the following global variables (where a `true`variable means that the corresponding item is broken):

```
var K_plumb, K_electr, K_floor, K_wall, K_clean: bool
```

Similar variables track the state of L(iving room), B(athroom), and (bed)R(oom). The activity state of the five contractors is tracked by the following global variables:

```
var plumb_active, electr_active, floor_active,
    wall_active, clean_active: bool
```

At the start, one supposes that there is no ongoing work in the room, meaning that the contractors are not working.

```
stateset initcond: ~K_busy & ~L_busy & ~B_busy & ~R_busy & ~plumb_active
         & ~electr_active & ~floor_active & ~wall_active & ~clean_active
```

After these declarations, we declare the modules. The module `Breaks` models plumbing and electrical failures. The code for this module is given in Figure 7. The body of the module contains a series of declarations of output transitions. As an example, the following transition models the fact that, when the plumbing in the kitchen is not broken (`~` means "not", and `K_plumb` tracks whether the kitchen plumbing works), then it can break, generating the output transition `break_K_plumb`, and signaling that the kitchen plumbing, floor, and walls need repair. Moreover, the room needs to be cleaned.

```
1  module Breaks:
2    stateless
3          K_plumb, K_electr, K_floor, K_wall, K_clean,
4          L_plumb, L_electr, L_floor, L_wall, L_clean,
5          B_plumb, B_electr, B_floor, B_wall, B_clean,
6          R_plumb, R_electr, R_floor, R_wall, R_clean
7
8    output break_K_plumb : { ~K_plumb ==> K_plumb' & K_floor' & K_wall'
         & K_clean' }
9    output break_L_plumb : { ~L_plumb ==> L_plumb' & L_floor' & L_wall'
         & L_clean' }
10   output break_B_plumb : { ~B_plumb ==> B_plumb' & B_floor' & B_wall'
         & B_clean' }
11   output break_R_plumb : { ~R_plumb ==> R_plumb' & R_floor' & R_wall'
         & R_clean' }
12
13   output break_K_electr : { ~K_electr ==> K_electr' & K_wall' &
         K_clean' }
14   output break_L_electr : { ~L_electr ==> L_electr' & L_wall' &
         L_clean' }
15   output break_B_electr : { ~B_electr ==> B_electr' & B_wall' &
         B_clean' }
16   output break_R_electr : { ~R_electr ==> R_electr' & R_wall' &
         R_clean' }
17 endmodule
```

Fig. 7. Module `Breaks` for the house example.

```
output break_K_plumb : { ~K_plumb ==> K_plumb' & K_floor' &
                         K_wall' & K_clean'}
```

All global variables are history free for this module.

The module `Calls` calls the repairmen and the cleaner when needed (the code of this module is given in Figure 8); as an example, the plumber is called using the following statement:

```
output call_plumb : { ~plumb_active &
       (K_plumb | L_plumb | B_plumb | R_plumb) ==> plumb_active' }
```

Note that all the variables are history free for this module. This choice is quite obvious since, as an example, there is no reason for `Calls` to track the value of `plumb_active` after it has called the plumber. If global variables where not history free, then one would be forced to add many new input rules to the module.

After the declaration of the modules `Breaks` and `Calls`, come the declarations of the modules for the five contractors.

The plumber, whose part of the code is given in Figure 9, and the other contractors keep track of whether they are working via a Boolean variable `working`. Also, they keep track of the room on which they are working via the local Boolean variables `Kw`, `Lw`, `Bw`, `Rw`. W hen called, the plumber is initially not working on any room.

```
1  module Calls:
2    stateless
3          K_plumb, K_electr, K_floor, K_wall, K_clean,
4          L_plumb, L_electr, L_floor, L_wall, L_clean,
5          B_plumb, B_electr, B_floor, B_wall, B_clean,
6          R_plumb, R_electr, R_floor, R_wall, R_clean,
7          plumb_active, electr_active, floor_active, wall_active,
               clean_active
8
9    output call_plumb : { ~plumb_active & (K_plumb | L_plumb | B_plumb |
          R_plumb ) ==> plumb_active' }
10   output call_electr : { ~electr_active & (K_electr | L_electr |
         B_electr | R_electr) ==> electr_active' }
11   output call_floor : { ~floor_active & (K_floor | L_floor | B_floor |
          R_floor ) ==> floor_active' }
12   output call_wall : { ~wall_active & (K_wall  | L_wall  | B_wall  |
         R_wall ) ==> wall_active' }
13   output call_clean : { ~clean_active & (K_clean | L_clean | B_clean |
          R_clean ) ==> clean_active' }
14
15 endmodule
```

Fig. 8. Module `Calls` for the house example. The module calls the repairmen and the cleaner.

```
input call_plumb : { local: ~plumb_active ==> working' := false }
```

When an active plumber, not working on any room, sees that the K(itchen) is unoccupied (`~K_busy`) and needs repair (`K_plumb`), the plumber starts to work in the K(itchen):

```
output K_start_plumb :
 { plumb_active & ~working & K_plumb & ~K_busy
   ==>
   working' & Kw' & K_busy' }
```

and similarly for the other rooms.

While working in the kitchen, the plumber does not expect anybody else to work in it. Thus, we have to define input transitions corresponding to the actions of the other contractors. As an example, the following rule forbids the electrician to start working in the kitchen if the plumber is still working there.

```
input  K_start_electr : { local: ~Kw ==>  }
```

One of the main drawbacks of this formalization is that we have to define many input transitions that differ only by their name but not by their contents. To simplify the declaration of such inputs, TICC allows the use of wildcard action names. Figure 9 shows how wildcard inputs can simplify the description of the module `Plumber`. Using the special character "`*`", input transition rules can be defined to match a set of actions instead of one action only. For instance, the pattern `K_*` on line 24 of Figure 9 matches any action whose name starts with `K_`.

```
1  module Plumber:
2    var working: bool
3    var Kw, Lw, Bw, Rw: bool
4    initial: ~working & ~Kw & ~Lw & ~Bw & ~Rw
5    stateless
6          K_plumb, K_electr, K_floor, K_wall, K_clean,
7          L_plumb, L_electr, L_floor, L_wall, L_clean,
8          B_plumb, B_electr, B_floor, B_wall, B_clean,
9          R_plumb, R_electr, R_floor, R_wall, R_clean
10
11   input call_plumb : {local: ~plumb_active ==> working' := false }
12   output done_plumb : { plumb_active & ~working & ~K_plumb & ~L_plumb
         & ~B_plumb & ~R_plumb ==> ~plumb_active' }
13
14
15   output K_start_plumb : { plumb_active & ~working & K_plumb & ~K_busy
          ==> working' & Kw' & K_busy' }
16   output L_start_plumb : { plumb_active & ~working & L_plumb & ~L_busy
          ==> working' & Lw' & L_busy' }
17   output B_start_plumb : { plumb_active & ~working & B_plumb & ~B_busy
          ==> working' & Bw' & B_busy' }
18   output R_start_plumb : { plumb_active & ~working & R_plumb & ~R_busy
          ==> working' & Rw' & R_busy' }
19   output K_done_plumb : { plumb_active & Kw ==> ~K_plumb' & ~Kw' & ~
         K_busy' & ~working' }
20   output L_done_plumb : { plumb_active & Lw ==> ~L_plumb' & ~Lw' & ~
         L_busy' & ~working' }
21   output B_done_plumb : { plumb_active & Bw ==> ~B_plumb' & ~Bw' & ~
         B_busy' & ~working' }
22   output R_done_plumb : { plumb_active & Rw ==> ~R_plumb' & ~Rw' & ~
         R_busy' & ~working' }
23
24   input K_* : { local: ~Kw ==> }
25   input L_* : { local: ~Lw ==> }
26   input B_* : { local: ~Bw ==> }
27   input R_* : { local: ~Rw ==> }
28 endmodule
```

Fig. 9. Module describing the plumber.

```
 1  open Ticc;;
 2
 3  parse "house.si";;
 4
 5  let breaks     = mk_sym "Breaks";;
 6  let calls      = mk_sym "Calls";;
 7  let plumber    = mk_sym "Plumber";;
 8  let electrician = mk_sym "Electrician";;
 9  let rudelectr  = mk_sym "RudeElectrician";;
10  let floors     = mk_sym "Floors";;
11  let walls      = mk_sym "Walls";;
12  let clean      = mk_sym "Clean";;
13
14  let c0 = compose breaks calls;;
15  let c1 = compose c0 plumber;;
16  let c2 = compose c1 electrician;;
17
18  let d2 = compose c1 rudelectr;;
```

Fig. 10. TICC script for the house example: `house.in`.

In module `Plumber`, variable `plumb_active` is a history variable, as the module plans to control its value. Variables `K_busy`, `L_busy`, `B_busy`, and `R_busy` are also history variables. This choice, combined with the declaration of the input transitions, ensures that the value of those variables can be changed by other modules only if the plumber is not working in the corresponding room.

We considered two different electrician modules. A "correct" implementation, `Electrician`, checks that the kitchen is free before starting to work in it:

```
output K_start_electr :
 { electr_active & ~working & K_electr & ~K_busy
   ==>
   working' & Kw' & K_busy' }
```

Note that above, the variable `Kw` is local to the electrician, and indicates whether the electrician is working on the kitchen; the equally-named variable `Kw` in (*) is instead local to the plumber. An "incorrect" implementation of the electrician, `WElectrician`, in the rush of getting things done, forgets to check whether somebody else is already at work in the kitchen:

```
output K_start_electr :
 { electr_active & ~working & K_electr ==> working' & Kw' & K_busy' }
```

TICC is able to detect that the composition of `Breaks`, `Calls`, `Plumber`, and `Electrician` is compatible (see lines from 14 to 16 of Figure 10), whereas it detects that the composition of `Breaks`, `Calls`, `Plumber`, and `WElectrician` is not. Thus, the protocol violation can be discovered before the complete system, consisting also of modules to repair floors and walls, is constructed. In fact, a simple check would have

revealed the problem already in the composition of `Plumber` and `WElectrician` (as computed in line 18 of Figure 10). When composing `Plumber` and `WElectrician`, TICC automatically synthesizes the assumption that (i) they are not both called to work, or (ii) no room needs to be repaired by both of them.

We also note that the protocol violation is revealed *thanks to the input assumption of the correct module* `Plumber`. In the game-based approach that underlies TICC, the input assumptions of correct modules constrain the protocol of modules that will be later composed into the system, preventing the composition of "rogue" modules. The verification of the correctness of interaction is simply a by-product of composition. This situation should be contrasted to the usual, non-game-based approach to modeling and verification. In the usual approach, detecting incompatibilities requires writing separate specifications of correctness, and can usually be performed only once all components are composed.

7 Additional Tool Features

While composition is certainly the most important operation that TICC can perform on modules, it is not the only one. This section is a brief introduction to the other features of the tool.

7.1 Symbolic Operations, Model Checking, and Simulation

A set of states, in TICC, can be defined via a formula specifying constraints on the values of the variables. TICC can parse such formulas, and construct a symbolic representation (an MDD) that enables it to manipulate the set. TICC can combine such sets with the usual Boolean operators, via the functions `set_or`, `set_and`, `set_implies`, and `set_not`; sets can also be compared using `set_is_subset` and `set_equal`. A set of states can be printed using the command `print_stateset` (printing is not optimized, and can lead to exponentially large printouts). TICC also contains an implementation of the classical CTL operators [9], allowing the user to verify properties of models via model checking. As usual, the CTL operators are documented in `doc/api/Ticc.html`.

Example 6. Consider the fire detection system given in Figure 2(a), and the script file in Figure 11. Line 11 builds the symbolic representation of a set ϕ consisting of the states where `ControlUnit.s = 2`, i.e., the firemen have been called. Line 13 prints the set of states that satisfy the CTL formula $\exists\Diamond\phi$, and line 15 prints the set of states that satisfy the CTL formula $\forall\Diamond\phi$.

TICC can also perform random simulation on symbolic modules, generating an HTML file with the result of the simulation. This is particularly useful in the early stages of model construction, to confirm that the model behaves as intended.

7.2 Closure

TICC allows the user to close a module with respect to the occurrence of input transitions. After several modules have been composed, the closure operation can be used

```
1  open Ticc;;
2  parse "fire-detector-disable.si";;
3
4  let fire1 = mk_sym "FireDetector1";;
5  let controlunit = mk_sym "ControlUnit";;
6  let comp = compose fire1 controlunit;;
7
8  let clone_fire1 = sym_clone fire1;;
9  simulate comp "Fire1.s = 0 & ControlUnit.s = 0", 5, "detector.html";;
10
11 let called_firemen = parse_stateset ("ControlUnit.s = 2");;
12 print_string "Can call the firemen:";;
13 print_stateset (ctl_e_f comp called_firemen);;
14 print_string "Always calls the firemen:";;
15 print_stateset (ctl_a_f comp called_firemen);;
```

Fig. 11. A script file illustrating individual operations.

to say that the environment is no longer able to provide a certain input. The following example illustrates the use of the closure operation in the context of CTL model checking.

Example 7. We consider a simple dining philosophers model, where n philosophers are sitting at a round table. Set between each pair of neighboring philosophers are n forks, so that all philosophers have a fork on their left, and one on their right. Each philosopher can either think or try to eat. To be able to eat, philosophers, being rather clumsy, have to use both forks on their sides.

Each philosopher `Phil` can be in one of 7 internal states that are enumerated with a local variable `s`. In `s=0`, `Phil` is thinking; a transition to `s=1` indicates the philosopher's desire for food. In state `s=4` the philosopher eats. To go from `s=1` to `s=4`, `Phil` has to grab the two forks. This can be done in any order (requiring the addition of two intermediate states `s=2` and `s=3`, depending on which fork has been chosen first). After having eaten, `Phil` releases the forks in nondeterministic order, and starts thinking again.

The TICC program of Figure 12 and its corresponding script file given in Figure 13 show an example of dining philosophers with $n = 2$ philosophers and thus 2 forks. The program can easily be extended to a greater number of philosophers. In the program, the philosophers are represented by modules `Phil1` and `Phil2`, while the forks with Boolean global variables `F1`, and `F2`, whose value is `true` if the fork is available, and `false` otherwise. The actions of grabbing and releasing forks are modeled by the actions `GrabF`x and `givebackF`x, where $x \in \{1,2\}$ identifies the fork. Since a fork is shared between two philosophers, each philosopher must both output these actions, and be able to accept them as input from other philosophers. This is the purpose of the wildcard input `input *`.

The problem is that, once `Phil1` and `Phil2` are composed, their composition can *still* accept the actions `GrabF`x and `givebackF`x from the environment. It is as if

```
1  var F1, F2: bool
2  stateset initcond: F1 & F2
3
4  module Phil1:
5        var s: [0..6]
6        initial: s = 0
7
8        input *: {}
9        local no_moves: { true ==> }
10       local wants_to_eat: { s = 0 ==> s' = 1 }
11       output grabF1: { s = 1 & F1 ==> s' = 2 & ~F1';
12             s = 3 & F1 ==> s' = 4 & ~F1' }
13       output grabF2: { s = 1 & F2 ==> s' = 3 & ~F2';
14             s = 2 & F2 ==> s' = 4 & ~F2' }
15       output givebackF1: { s = 4 ==> s' = 5 & F1';
16             s = 6 ==> s' = 0 & F1' }
17       output givebackF2: { s = 4 ==> s' = 6 & F2';
18             s = 5 ==> s' = 0 & F2' }
19 endmodule
20
21 module Phil2:
22       var s: [0..6]
23       initial: s = 0
24
25       input *: {}
26       local no_moves: { true ==> }
27       local wants_to_eat: { s = 0 ==> s' = 1 }
28       output grabF2: { s = 1 & F2 ==> s' = 2 & ~F2';
29             s = 3 & F2 ==> s' = 4 & ~F2' }
30       output grabF1: { s = 1 & F1 ==> s' = 3 & ~F1';
31             s = 2 & F1 ==> s' = 4 & ~F1' }
32       output givebackF2: { s = 4 ==> s' = 5 & F2';
33             s = 6 ==> s' = 0 & F2' }
34       output givebackF1: { s = 4 ==> s' = 6 & F1';
35             s = 5 ==> s' = 0 & F1' }
36 endmodule
```

Fig. 12. A TICC dining philosophers model: `dining.si`.

```
 1  open Ticc;;
 2
 3  parse "phil.si";;
 4
 5  let phil1 = mk_sym "Phil1";;
 6  let phil2 = mk_sym "Phil2";;
 7  let comp_phils = compose phil1 phil2;;
 8
 9  let initial = parse_stateset "Phil1.s = 0 & Phil2.s = 0 & F1 & F2 ";;
10  let bad_fork = parse_stateset "Phil1.s = 0 & Phil2.s = 0 & ~F2";;
11
12  let can_reach_bad_fork_exists = ctl_e_f comp_phils bad_fork;;
13  let result = set_and can_reach_bad_fork_exists initial;;
14  print_stateset result;;
15
16  let comp_phils_close = close comp_phils "*";;
17
18  let can_reach_bad_fork_exists = ctl_e_f comp_phils_close bad_fork;;
19  let result = set_and can_reach_bad_fork_exists initial;;
20  print_stateset result;;
```

Fig. 13. A TICC script for the dining philosophers.

passers-by were allowed to pick up and put down forks! Indeed, in the composition of `Phil1` and `Phil2`, we can start from the state where `Phil1` and `Phil2` are both thinking and `F2` is available and reach a state where the philosophers are still thinking but `F2` is not available, as it has been "picked up" by the environment. This is shown by the fact that the stateset printed at line 14 is not empty.

This clearly does not make sense: once `Phil1` and `Phil2` are composed, we should be able to say that the forks are no longer in the environment's reach. To this end, we *close* the composition of `Phil1` and `Phil2` with respect to all input actions.[8] Once this is done, the state where both philosophers are thinking but `F2` is not available is no longer reachable, and indeed the printout from line 20 is the empty set (represented as `(0)`).

8 Conclusions

Interface theories are the subject of many recent works. The sociable interface model presented in this paper is only one of them. Interface models that appeared before sociable interfaces include interface automata [13, 15] and interface modules [14, 8]. Those models were based on a communication with either actions, or variables, but not both.

Sociable interfaces do not break new ground in the conceptual theory of interface models. However, by allowing both actions and variables in the communication process, they take advantage of the existing models and provide rich communication primitives.

[8] In general, we can close a module with respect to any set of actions.

The tool TICC is certainly not the first tool that implements an interface model, and even not the most complete. As an example, the tool CHIC that implements a synchronous, variable-based interface theory is able to handle pushdown games while TICC cannot.

However, one major difference between TICC and its predecessors is its ability to use rich communication primitives to model components in a very compact and natural way. Another strong point of the tool is its symbolic implementation which makes it very efficient and easily extensible.

TICC is a tool in constant evolution, and so is the sociable interface model. As an example, we are currently developing a real-time extension of the tool, based on the *Timed Interfaces* of [16]. This is a large and complex endeavor, as the game-theoretic machinery of TICC will have to be replaced with one suited to real-time games. Another direction we are considering is the implementation of the alternating-time temporal logic of [2]. This logic is more suitable to model check open systems than CTL.

References

1. B. Adler, L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, V. Raman, and P. Roy. Ticc, a tool for interface compatibility and composition. In *Proceedings 18th International Conference on Computer Aided Verifi cation (CAV)*, volume 4144 of *Lecture Notes in Computer Science*. Springer, 2006. to appear.
2. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symp. Found. of Comp. Sci.*, pages 100–109. IEEE Computer Society Press, 1997.
3. R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: modularity in model checking. In *CAV 98: Proc. of 10th Conf. on Computer Aided Verifi cation*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 521–525. Springer-Verlag, 1998.
4. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
5. A. Chackrabarti, L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Resource interfaces. In *EMSOFT 03: 3rd Intl. Workshop on Embedded Software*, volume 2855 of *Lect. Notes in Comp. Sci.*, pages 117–133. Springer-Verlag, 2003.
6. A. Chackrabarti, L. de Alfaro, M. Jurdziński, K. Chatterjee, T.A. Henzinger, and F.Y.C. Mang. CHIC: Checker for interface compatibility, 2003. www-cad.eecs.berkeley.edu/ tah/chic/.
7. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, Marcin Jurdziński, and F.Y.C. Mang. Interface compatibility checking for software modules. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verifi cation*, volume 2404 of *Lect. Notes in Comp. Sci.*, pages 428–441. Springer-Verlag, 2002.
8. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verifi cation*, volume 2404 of *Lect. Notes in Comp. Sci.*, pages 414–427. Springer-Verlag, 2002.
9. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.
10. L. de Alfaro. Game models for open systems. In *Proceedings of the International Symposium on Verifi cation (Theory in Practice)*, volume 2772 of *Lect. Notes in Comp. Sci*. Springer-Verlag, 2003.
11. L. de Alfaro, R. Alur, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Meyer-Kirsch, and B.Y. Wang. Mocha: A model checking tool that exploits design structure. In *ICSE 01: Proceedings of the 23rd International Conference on Software Engineering*, 2001.

12. L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, P. Roy, and M. Sorea. Sociable interfaces. In *Proceedings of 5th International Workshop on Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Computer Science*, pages 81–105. Springer, 2005.
13. L. de Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM Press, 2001.
14. L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In *EMSOFT 01: 1st Intl. Workshop on Embedded Software*, volume 2211 of *Lect. Notes in Comp. Sci.*, pages 148–165. Springer-Verlag, 2001.
15. L. de Alfaro and T.A. Henzinger. Interface-based design. In *Engineering Theories of Software Intensive Systems, proceedings of the Marktoberdorf Summer School*. Kluwer, 2004.
16. L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Timed interfaces. In *Proceedings of the Second International Workshop on Embedded Software (EMSOFT 2002)*, Lect. Notes in Comp. Sci., pages 108–122. Springer-Verlag, 2002.
17. L. de Alfaro and M. Stoelinga. Interfaces: A game-theoretic framework to reason about open systems. In *FOCLASA 03: Proceedings of the 2nd International Workshop on Foundations of Coordination Languages and Software Architectures*, 2003.
18. M. Faella and A. Legay. Some models and tools for open systems. Technical report, University of Santa Cruz, 2005. Proceedings of FIT05.
19. E. A. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspect of Computing Journal*, 2003.
20. Xavier Leroy. Objective caml. http://caml.inria.fr/ocaml/index.en.html.
21. N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.
22. Fabio Somenzi. Cudd: Cu decision diagram package. http://vlsi.colorado.edu/ fabio/CUDD/cuddIntro.html.
23. A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *Proceedings International Conference CAD (ICCAD-91)*, 1990.

An Operator-based Approach to Incremental Development of Conform Protocol State Machines

Arnaud Lanoix, Dieu-Donné Okalas Ossami and Jeanine Souquières

LORIA – CNRS – Université Nancy 2
Campus scientifique
F-54506 Vandoeuvre-Lès-Nancy
{lanoix,okalas,souquier}@loria.fr

Abstract. An incremental development framework which supports a conform construction of Protocol State Machines (PSMs) is presented. We capture design concepts and strategies of PSM construction by sequentially applying some development operators: each operator makes evolve the current PSM to another one. To ensure a conform construction, we introduce three conformance relations, inspired by the specification refinement and specification matchings supported by formal methods. Conformance relations preserve some global behavioral properties. Our purpose is illustrated by some development steps of the card service interface of an electronic purse: for each step, we introduce the idea of the development, we propose an operator and we give the new specification state obtained by the application of this operator and the property of this state relatively to the previous one in terms of conformance relation.

Keywords. protocol state machine, incremental development, development operator, exact conformance, plugin conformance, partial conformance

1 Introduction

Software design is an incremental process where modifications of the system's functionalities can occur at every stage of the development. In order to increase the software quality, it is important to understand the impact of these modifications in terms of lost, added or changed global behaviors.

UML 2.0 [1] introduces protocol state machines (PSMs) to describe valid sequences of operation calls of an object. PSMs are a specialization of generic UML state machines without actions nor activities. Generic state machines are based on the widely recognized statechart notations introduced by Harel [2].

In protocol state machines, transitions are specified in terms of pre/post conditions and state invariants can be given. PSMs are used for developing behavioral abstractions of complex, reactive software. Typically, these state machines provide precise descriptions of component behavior and can be used – combined

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/695

with a refinement process – for generating implementations. This framework provides a convenient way to model the ordering of operations on a classifier. Notice that the literature about PSMs is quite poor [3,4].

The notion of conformance of PSMs is an important issue for the development. It is considered in UML 2.0, but limited to explicitly declaring, via the protocol conformance model element, that a specific state machine "conforms" to a general PSM. The definition given in [1] remains very general and does not ease its use in practice.

The conformance between development steps has been studied in formal specification approaches. For example, the B method proposes a refinement mechanism [5,6,7]: a system development begins by the definition of an abstract view which can be refined step by step until an implementation is reached. In the framework of algebraic specifications, this notion of conformance has been studied and has given several specification matchings [8]. Meyer and Santen propose a verification of the behavioral conformance between UML and B [9].

This notion is also very important in the field of test. In this domain, conformance is usually defined as testing to see if an implementation faithfully meets the requirements of a standard or a specification. Conformance testing means the use of conformance relations, like the *conf* or *ioco* relations [10], based on Labeled Transition Systems (LTS) or process algebras. Other notions of conformance in the context of LTS are the equivalence relations [11], (bi)simulations [12,13] and refinement [14,15].

Some notions of conformance have been taken into account for the statecharts [2] or UML 1.x state diagrams. The equivalence of state machines has been studied in [16], the conformance testing in [17] and some refinements in [18,19,20]. The majority of these works are based on a semantics of state machines given in terms of LTS using extended hierarchical automata [21,22,23].

The idea of following an incremental construction is not new and has been addressed in several works. Some propositions for the incremental design of a part of the statechart specifications are discussed in [24,4]. An operator-based framework to the incremental development of multi-view UML and B specifications is defined in [25].

This work deals with the incremental development process of PSMs, and, in particular, with the expression of the property between two development steps by means of the conformance relations. Based on formal specification matchings and refinement, we propose three conformance relations, called ExactConformance, PluginConformance and PartialConformance expressing three levels of the preservation of the behavior. In order to help a conform step-by-step construction process, we propose development operators. In [26], we have introduce some operators to deal with subPSMs. This paper extends the approach proposed in [26] by providing other development operators to refine a PSM thanks to the modifications performed on its associated interface.

The paper is structured as follows. Section 2 introduces our running case study and presents UML 2.0 protocol state machines. After a presentation of the UML 2.0 PSM redefinition, Section 3 gives three conformance relations, namely

exact, plugin and partial conformances. Section 4 presents some development steps of the case study; for each step we introduce the idea of the development, we propose an operator, we give the new specification state and the property of this state relatively to the previous one in terms of conformance. Section 5 concludes and gives some perspectives.

2 Protocol state machines

This section introduces the UML protocol state machines and the example used throughout this paper.

2.1 Case study: CEPS card

We consider as running example, a part of the Common Electronic Purse Specifications (CEPS) [27]. The system is based on an infrastructure of terminals on which a customer can pay for goods, using a payment card which stores a certain - reloadable - amount of money. In the sequel, we will focus on the card application.

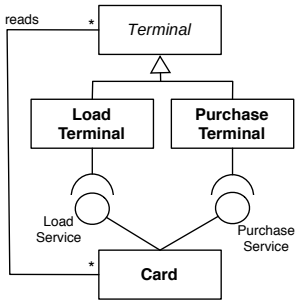


Fig. 1. CEPS architecture

Figure 1 shows the architecture of the system: Card represents a payment card while LoadTerminal and PurchaseTerminal represent respectively terminals used to reload the card and terminals used for purchases. Card provides the PurchaseService and LoadService interfaces to communicate with the respective terminals.

2.2 UML 2.0 protocol state machines

PSMs are introduced in UML 2.0 [1] as state machine variants defined in the context of a classifier (interface or class) to model the order of operations calls. PSMs differ from generic state machines by the following restrictions:

– States cannot show entry actions, exit actions, internal actions, or do activities.

- State invariants can be specified.
- Pseudostates cannot be deep or shadow history kinds.
- Transitions cannot show effect actions or send events as generic state machines can.
- Transitions have pre and post-conditions; they can be associated to operation calls.

A PSM may contain one or more regions which involve vertices and transitions. A protocol transition connects a source vertex to a target vertex. A vertex is either a pseudostate or a state with incoming and outgoing transitions. States may contain zero or more regions.

- Pseudostates can be *initial*, *entry point*, *exit point* or *choice* kinds; a choice pseudostate realizes a conditional branch.
- A state without region is a *simple* state; a *final* state is a specialization of a state representing the completion of a region.
- A state containing one or more regions is a *composite* state that provides a hierarchical group of (sub)states; a state containing more than one region is an *orthogonal* state that models a concurrent execution.
- A *submachine* state is semantically equivalent to a composite state. It refers to a submachine (subPSM) where its regions are the regions of the composite state.

2.3 Example: PurchasePSM

In the sequel, we focus on the PurchaseService interface and its associated PSM PurchasePSM given Figure 2. The interface PurchaseService provides an attribute, balance, which represents the amount of money available on the card. The PSM PurchasePSM describes the following behavior: its initial state is Ready. First, the purchase terminal, used to read the card, is authentified and the TerminalAccepted state is reached. Next, the PSM reaches the PurchaseRealized state if there is enough money on the card, which is ensured by the precondition [balance > 0].

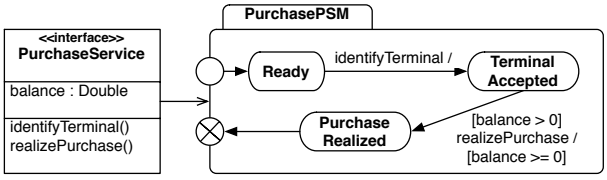


Fig. 2. PurchasePSM

3 Conformance relations

The protocol conformance relation [1] is used to explicitly declare that a specific state machine conforms to a general PSM. The given semantics is the preservation of pre/post conditions and state invariants of the general PSM in the more specific one. For our point of view, the definition of the protocol conformance relation remains too very general to be used in practice and does not allow the designer how to decide on conformance between two PSMs.

State machine redefinition is also considered in UML 2.0. A specialized state machine is an extension of a general state machine where regions, vertices and transitions have been added or redefined. So, it has additional elements.

A simple state can be redefined to a composite state by adding one or more regions. A composite state can be redefined by either extending its regions or by adding regions as well as by adding entry and exit points. A region can be extended by adding vertices and transitions and by redefining states and transitions. A submachine state may be redefined by another submachine state that provides the same entry/exit points and adds new entry/exit points.

Let PSM_1 and PSM_2 be a PSM and another PSM obtained by a transformation of PSM_1 by performing a development step. In order to study the construction-based conformance between PSM_1 and PSM_2, we introduce three relations. These relations describe different levels of behavioral preservations corresponding to properties of the new PSM relatively to the previous one.

1. PluginConformance: $\mathsf{PSM}_2 \sqsubseteq \mathsf{PSM}_1$.
   We have a PluginConformance relation between PSM_2 and PSM when PSM_2 provides all the functionalities of PSM_1 and when the new functionalities provided by PSM_2 don't conflict with the ones of PSM_1. We are able to "plugin" PSM_2 for PSM_1.
2. PartialConformance: $\mathsf{PSM}_2 \sqsupseteq \mathsf{PSM}_1$.
   The PartialConformance relation is the reciprocal relation of the PluginConformance relation: $\mathsf{PSM}_2 \sqsupseteq \mathsf{PSM}_1$ iff $\mathsf{PSM}_1 \sqsubseteq \mathsf{PSM}_2$. In other words, this relation occurs between PSM_2 and PSM_1 when PSM_2 provides less functionalities than PSM_1, but all the functionalities provided by PSM_2 are provided by PSM_1.
3. ExactConformance: $\mathsf{PSM}_2 \equiv \mathsf{PSM}_1$.
   We have an ExactConformance relation between PSM_2 and PSM_1 if the two PSMs are equivalent and completely interchangeable. All Observable functionalities provided by PSM_1 and by PSM_2 must be the same. The ExactConformance relation is symmetric.
   The ExactConformance relation is a specialization of both PluginConformance and PartialConformance relations; we can easily demonstrate that if $\mathsf{PSM}_2 \equiv \mathsf{PSM}_1$ then $\mathsf{PSM}_2 \sqsubseteq \mathsf{PSM}_1$ and $\mathsf{PSM}_2 \sqsupseteq \mathsf{PSM}_1$.

Notice that the ExactConformance relation is a strong requirements often incompatible with a construction process. Sometimes a weaker match as PluginConformance or PartialConformance can be enough.

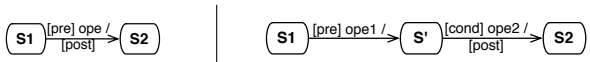
There is no formal definitions of the previous relations in this paper. Interested reader might find some proposals in [16,17,19,20]. We focus on their uses to guid an incremental developement.

4 Conform development

Let us see some development steps of the case study, starting from PurchasePSM and its associated interface PurchaseService, presented Figure 2. Our objective is to elaborate from this state a more complete PSM that presents the functionalities provided by the card following the interface modifications. For each step, we give the general idea of the evolution involved which respects to the new associated interface, the development operator which is applied on the current state and the conformance property that is preserved, which is the properties of the new state relatively to the previous one.

4.1 Introducing sequences of operations

Figure 2 gives an abstraction of the authentication process. The operation identifyTerminal() can be decomposed by the sequence of operations readCertificate(term_id), followed by acceptTerminal().



(a) before (b) after

Fig. 3. refine_by_sequences()

This sequence is formally described by an UML annotation. The syntax used is the following:
ope() := ope1() ; [cond] ope2()
that expresses the substitution of ope() by ope1() followed by ope2() under the condition [cond] (see Figure 3).

We define a construction operator refine_by_sequences() which substitutes the considered transition by the sequence of new transitions as shown Figure 3. If [cond] is defined, then PartialConformance is preserved by this operator; otherwise, ExactConformance is preserved.

The PSM PurchasePSM_2, given Figure 4, corresponds to the application of the operator refine_by_sequences() on the transition identifyTerminal which substitutes identifyTerminal by readCertificate(term_id) and acceptTerminal. Figure 4 shows also the modifications of the interface associated to PurchasePSM. A new attribute card_id is added to authenticate a terminal by exchange of certificates[1].

[1] Notice that PurchaseService_2 interface shows only the updated informations of PurchaseService.

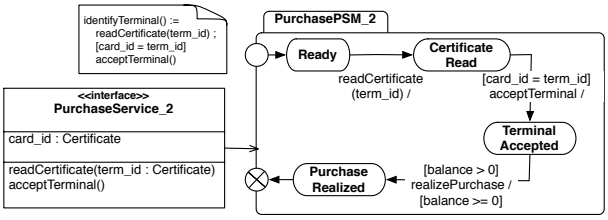


Fig. 4. PurchasePSM_2

4.2 Introducing complementary behaviors

When looking at the transition acceptTerminal between the states CertificateRead and TerminalAccepted on Figure 4, we remark that all the possible cases are not considered. The case where a valid terminal certificate is read, expressed by the precondition [card_id = term_id], is the only one to be taken into account. What happens when term_id is not a valid certificate? This new requirements involves the introduction of a new transition and a new state.

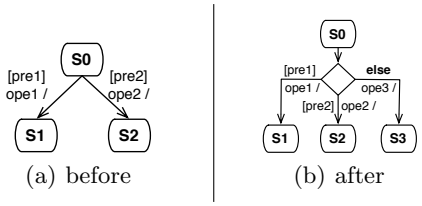


Fig. 5. complement_transition()

The operator complement_transition() proposes to introduce from a selected vertex and its outgoing transitions, a (default) complementary transition by using a choice pseudostate as shown Figure 5. Since the operator complement_transition() adds new functionalities, PluginConformance is preserved.

Applying the complementary_transition() operator on the state CertificateRead leads to a new PSM PurchasePSM_3 shown Figure 6. A choice pseudostate and a new state TerminalRefused are introduced.

Figure 7, a new exit point is introduced jointly with a transition from the TerminalRefused state to the new exit point using basic construction operators add_vertex() and add_transition() defined in [26]. Then, PluginConformance is preserved.

4.3 Reusing refine_by_sequences()

Let us consider now the transition realizePurchase between TerminalAccepted and PurchaseRealized states. We want to decompose this transition into two succes-

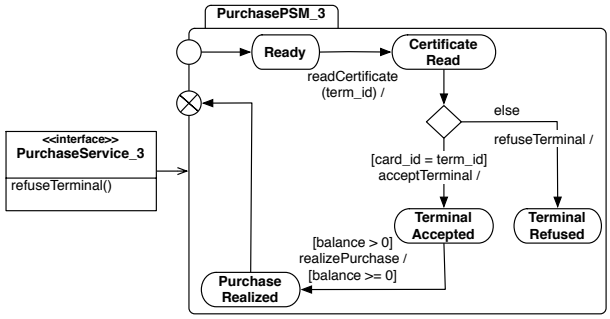


Fig. 6. PurchasePSM_3

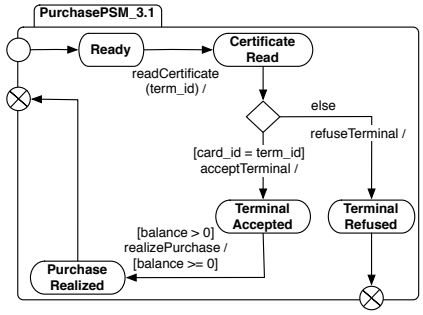


Fig. 7. PurchasePSM_3.1

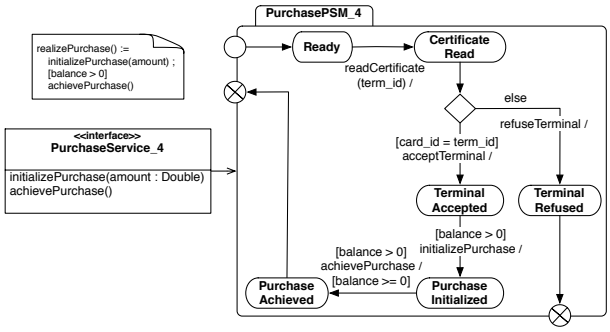


Fig. 8. PurchasePSM_4

sive transitions initializePurchase(amount) and achievePurchase to describe more precisely the purchase functionality.

The previous operator refine_by_sequences() is applied again to obtain a new PSM PurchasePSM_4 given Figure 8.

4.4 Introducing conditional behaviors

In the current development state, the achievePurchase transition is still abstract. It corresponds to two (conditional) behaviors: if there is enough money on the card to pay the purchase, then the purchase is realized and the balance is debited. Otherwise, the purchase must be canceled.

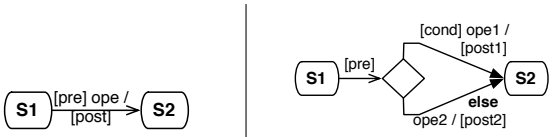


Fig. 9. refine_by_conditions()

A construction operator refine_by_conditions() is defined to substitute the considered transition by a conditional behavior expressed by an UML annotation which respects the following syntax:
ope() := **if** [cond] **then** ope1() [post1] **else** ope2() [post2]
Figure 9 illustrates this operator. It preserves the ExactConformance when the following obligation proofs are satisfied:

- (pre**@pre and** cond**@pre and** post1) **implies** post
- (pre**@pre and not** cond**@pre and** post2) **implies** post

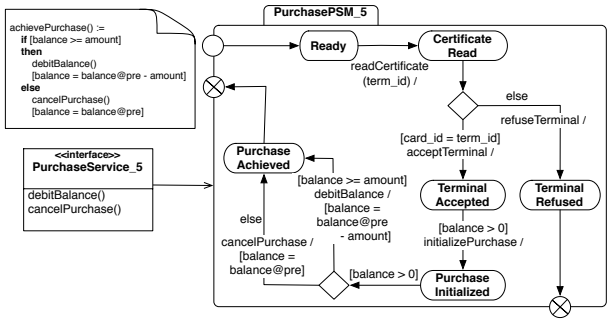


Fig. 10. PurchasePSM_5

The application of refine_by_conditions() on achievePurchase gives the new PSM PurchasePSM_5 by substituting the achievePurchase transition by debitBalance and cancelPurchase (see figure 10).

Since (balance**@pre** > 0 **and** balance**@pre** >= amount **and** balance = balance**@pre** - amount) **implies** (balance >= 0), and, (balance**@pre** > 0 **and** balance**@pre** < amount **and** balance = balance**@pre**) **implies** (balance > 0) are satisfied, we conclude that ExactConformance is preserved.

4.5 Splitting states

We can observe in PurchasePSM_5 that the two transitions debitBalance and cancelPurchase reach the same state PurchaseAchieved. Nevertheless, they describe different behaviors. We want to split PurchaseAchieved into two different states BalanceDebited and PurchaseCanceled to illustrate the difference.

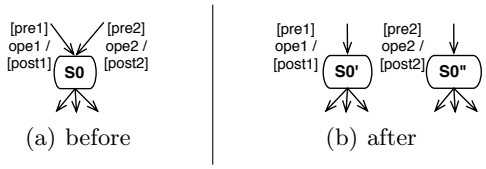


Fig. 11. split_state()

The construction operator split_state() depicted Figure 11 considers a vertex and its incoming transitions. For each incoming transition, the vertex is duplicated. All the outgoing transitions are also duplicated. Since this construction operator only duplicates behaviors, it preserves ExactConformance.

The application of this operator to the state PurchaseAchieved gives two new states BalanceDebited and PurchaseCanceled as shown Figure 12.

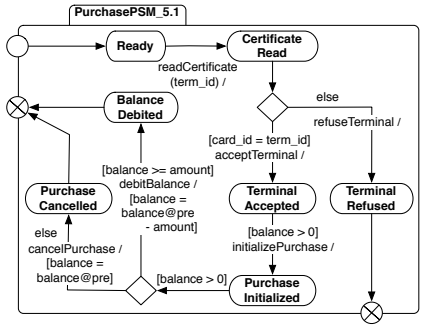


Fig. 12. PurchasePSM_5.1

When applying once again the split_state() operator to the exit pseudostate, we obtain the PSM PurchasePSM_5.2 given Figure 13.

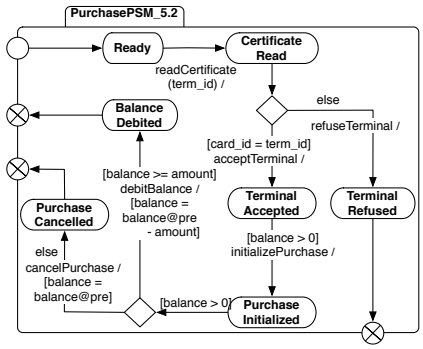


Fig. 13. PurchasePSM_5.2

An overview of a part of the followed development process is given Figure 14. Each development state is composed of a PSM and its associated interface and transitions between development states express the application of a development operators and the properties between two states: Refinement for interfaces and Conformance for PSMs.

5 Conclusion and future work

Specifying complex systems is a difficult task which cannot be done in one step. In a typical design process, the designer starts with a first draft model and transforms it by a step-by-step process into a more and more complex model.

The design approach we propose in this paper uses a set of construction operators to make evolve protocol state machines preserving behavioral properties. Three Conformance relations ExactConformance, PluginConformance and PartialConformance have been defined. The use of these operators has been illustrated on the development of a part of the CEPS case study.

Further work will focus on a generalization of our step-by-step construction of PSM by studying other construction operators, like operators for removing elements. We are currently exploring other particularities of PSMs like state invariants and transition post-conditions.

We also consider the formalization of the definition of the Conformance relations ExactConformance, PluginConformance and PartialConformance inspired by results in formal methods like refinement [7] and specification matchings [8]. The verification of the conform development can be done by translating the obtained PSM into a tool-supported language such that B [28,29] or TLA [30,31].

Another perspective concerns the implementation of a tool to assist in the development of PSMs based on our construction operators.

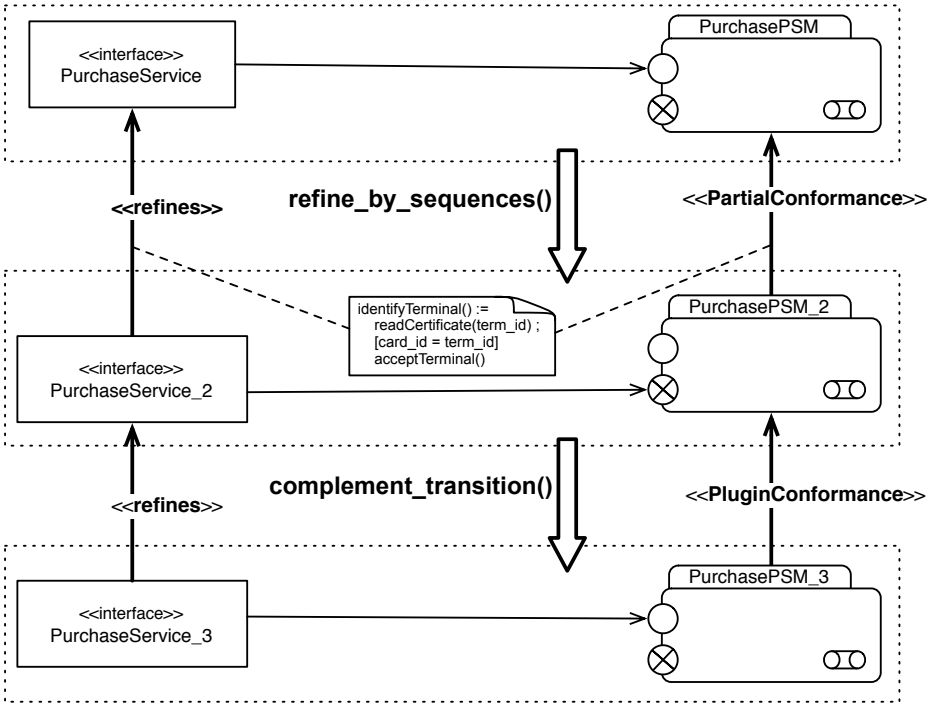


Fig. 14. Incremental development of PurchasePSM

References

1. Object Management Group: UML superstructure specification, v2.0 (2005)
2. Harel, D.: Modeling Reactive Systems With Statecharts. Mac Graw Hill (1998)
3. Mencl, V.: Specifying component behavior with port state machines. ENTCS **101C** (2004) 129–153
4. Gout, O., Lambolais, T.: UML Protocol State Machines Incremental Construction: a Conformance-based Refinement Approach. Research Report RR05/027, LGI2P (2005)
5. Morris, J.M.: A theoretical basis for stepwise refinement and programming calculus. Science of Computer Programming **9** (1987) 287–306
6. Back, R.J.: A calculus of refinements for program derivations. Acta Informatica (1988) 593–624
7. Abrial, J.R.: The B Book. Cambridge University Press (1996)
8. Zaremski, A.M., Wing, J.M.: Specification matching of software components. ACM Transaction on Software Engeniering Methodolology **6** (1997) 333–369
9. Meyer, E., Santen, T.: Behavioral Conformance Verification in an Integrated Approach Using UML and B. In: (IFM00), Integrated Formal Methods. Volume 1945 of LNCS., Springer Verlag (2000) 358
10. Tretmans, J.: Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. Computer Networks and ISDN Systems **29** (1996) 49–79
11. De Nicola, R.: Extensional equivalences for transition systems. Acta Informatica **24** (1987) 211–237
12. Milner, R.: Communication and concurrency. Prentice-Hall, Inc. (1989)
13. Fernandez, J.C.: An implementation of an efficient algorithm for bisimulation equivalence. Science of Computer Programming **13** (1990) 219–236
14. Bellegarde, F., Julliand, J., Kouchnarenko, O.: Ready-simulation is not ready to express a modular refinement relation. In: Fundamental Aspects of Software Engineering (FASE'00). Volume 1783 of LNCS., Springer Verlag (2000) 266–283
15. Kouchnarenko, O., Lanoix, A.: Refinement and verification of synchronized component-based systems. In Araki, K., Gnesi, S., D., M., eds.: Formal Methods (FM'03). Volume 2805 of LNCS., Springer Verlag (2003) 341–358
16. Maggiolo-Schettini, A., Peron, A., Tini, S.: Equivalences of statecharts. In: Proc. of the 7th Int. Conf. On Concurrency Theory (CONCUR'96), Springer-Verlag (1996) 687–702
17. Latella, D., Massink, M.: On testing and conformance relations of UML statechart diagrams behaviours. In ACM, ed.: Int. Symposium on Software Testing and Analysis. (2002)
18. Al'Achhab, M.: Specification and verification of hierarchical systems by refinement. In: Modelling and Verifying Parallel Processes (MOVEP'04). (2004)
19. Meng, S., Naixiao, Z., Barbosa, L.S.: On semantics and refinement of UML statecharts: A coalgebraic view. In: Proc. of the 2nd In. Conf. on Software Engineering and Formal Methods (SEFM'04). (2004)
20. Knapp, A., Merz, S., Wirsing, M., Zappe, J.: Specification and refinement of mobile systems in MTLA and mobile UML. Theoretical Computer Science (2005)
21. Mikk, E., Lakhnech, Y., Siegel, M.: Hierarchical automata as model for statecharts. In: Third Asian Computing Science Conference on Advances in Computing Science (ASIAN'97), London, UK, Springer Verlag (1997) 181–196

22. Latella, D., Majzik, I., Massink, M.: Towards a formal operational semantics of UML statechart diagrams. In: 3rd Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), Kluwer (1999) 331–347
23. Von der Beeck, M.: Formalization of UML-Statecharts. In: UML'01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, Springer-Verlag (2001) 406–421
24. Scholz, P.: Incremental design of statechart specifications. Science of Computer Programming **40** (2001) 119–145
25. Okalas Ossami, D.D., Souquières, J., Jacquot, J.P.: Consistency in UML and B multi-view specifications. In: Proc. of the Int. Conf. on Integrated Formal Methods, IFM'05. Number 3771 in LNCS, Springer-Verlag (2005) 386–405
26. Lanoix, A., Souquières, J.: A step-by-step process to build conform UML protocol state machines. Research Report ccsd-00019314, LORIA (2006) http://hal.ccsd.cnrs.fr/ccsd-00019314.
27. CEPSCO: Common electronic purse specifications, functional requirements, v6.3 (1999)
28. Ledang, H., Souquières, J.: Contributions for modelling UML state-charts in B. In: Third International Conference on Integrated Formal Methods - IFM'2002, Turku, Finland (2002)
29. Sekerinski, E., Zurob, R.: Translating statecharts to b. In: IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods, London, UK, Springer-Verlag (2002) 128–144
30. Deiss, T.: An Approach to the Combination of Formal Description Techniques: Statecharts and TLA. In: 1st International Conference on Integrated Formal Methods, IFM'99, Springer (1999) 231–250
31. Freinkel, C.: An Approach to Combining UML and TLA+ in Software Specification. Technical reports, University of Nevada, Reno (2003)

Extended Abstract: Explaining Data Type Reduction in the Shape Analysis Framework

Björn Wachter

Saarland University,
Im Stadtwald,
Saarbrücken, Germany
bwachter@cs.uni-sb.de

Abstract. Automatic formal verification of systems composed of a large or even unbounded number of components is difficult as the state space of these systems is prohibitively large. Abstraction techniques automatically construct finite approximations of infinite-state systems from which safe information about the original system can be inferred. We study two abstraction techniques shape analysis, a technique from program analysis, and data type reduction, originating from model checking. Until recently we did not properly understand how shape analysis and data type reduction relate. We shed light on this relation in a comprehensive way. This is a step towards a more unified view of abstraction employed in the static analysis and model checking community.

1 Introduction

We consider analysis techniques for parameterized systems such as protocols where the number of participating processes is a parameter These models are composed of processes that run in a parallel, interleaved fashion. The state of the model consists of the local states of all constituent processes. Typically one wants to verify first-order temporal properties, i.e. safety properties such as mutual exclusion and liveness properties such as lack of starvation.

Finitary abstraction techniques generate a finite state model that approximates the original infinite state model preserving certain properties. A finitary abstraction technique has typically two constituents (1) a state abstraction function that maps states of the original model to states of the abstract model and (2) a method to compute transitions between abstract states, i.e. the behavior of the abstract model. The finite state model is subject to reachability analysis or to a finite-state model checker. Several finitary abstractions have been proposed, such as counter abstraction[PXZ], canonical abstraction [SRW02,Yah01] and data type reduction [McM00,DW03].

In previous work [Wac05], we have studied a model checking framework for parameterized systems based on canonical abstraction that lends ideas from data type reduction. Notably, data type reduction can be expressed in the same framework which is the topic of this work.

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/701

1.1 State Abstractions

Predicate abstraction. Predicate abstraction approximates the state of a program by a tuple of Boolean values that record if certain properties hold or not. For example, instead of storing an integer variable x one only keeps track of whether or not $x > 0$ holds. Predicate abstraction has been successfully applied to sequential programs.

Running Example. To demonstrate the abstractions, we consider as an example a parameterized system in which each process p has a program counter $\texttt{PC}(p)$ giving the process' current control location; a control location is a member of the set $\{a, b, c, d\}$. The example state consists of 9 processes.

Counter Abstraction. Counter abstraction [PXZ] assumes that processes are finite state, i.e. there exists a finite set Σ of local states. For each local state $\sigma \in \Sigma$, a counter variable C_σ is used that records the number of processes currently in state σ. To obtain a finite abstract domain the counters are typically cut off at two. An abstract state is a mapping $C : \Sigma \to \{0, 1, \geq 2\}$. As the size of the abstract state space is exponential in the size of Σ, counter abstraction falls short of infinite or very large local state spaces.

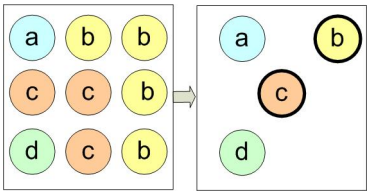


Fig. 1. Counter abstraction.

Figure 1 shows, left to the arrow, a concrete state with 9 processes that is abstracted to the abstract state right of the arrow. The circles denote concrete processes and the letters in the circles the value of the program counter. Note that in this example the set of local states is $\Sigma = \{a, b, c, d\}$. The abstract state has four counters one for each element of Σ. We think of the non-zero counters as abstract processes, as they stand for concrete processes. We symbolize each abstract process by a circle with a thin border if the counter is one, and by a circle with a thick border if the counter has at least value 2.

Canonical abstraction. As opposed to counter abstraction, canonical abstraction is applicable to systems where the local state space is infinite. Intuitively, canonical abstraction first abstracts local state per process, then processes with the same *abstract* local state are collapsed to one abstract process similar to counter abstraction. Local state is abstracted to a vector in which each position encodes the truth of a predicate ranging over processes. Predicates have defining formulas that may refer to local and global state, informally stated predicates can refer to the environment of a process.

Canonical abstraction admit predicates ranging over pairs of processes. For the sake of brevity, we omit these aspects of canonical abstraction for now.

Returning to the running example, we define two predicate: one predicate $\mathtt{at}_a(p)$ is true of a process if it is in control location a, $\mathtt{at}_a(p) \equiv \mathtt{PC}(p) = a$, the other predicate $\mathtt{at}_b$ holds for a process that is is control location b, $\mathtt{at}_b \equiv \mathtt{PC}(p) = b$. Figure 2 depicts the concrete state and its canonical abstraction. Abstract processes are two-component boolean vectors where the first components stands for the truth of predicate $\mathtt{at}_a$ and the second component for $\mathtt{at}_b$. The process in location a is mapped to the abstract process $(1, 0)$, the one of the processes in location b is $(0, 1)$ all other processes have $(0, 0)$.

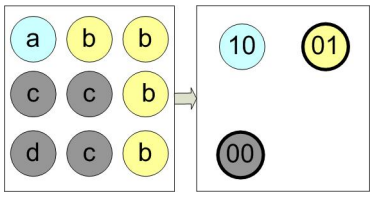


Fig. 2. Canonical abstraction

1.2 The Migration Problem

The previously described abstractions are sufficient to verify and infer invariants, yet, let alone, too coarse to verify first-order properties. For example, they would not allow us to check if every process will eventually reach location b. Consider the process in Figure 2 that is at control location a. Let us assume it moves on to location b. In an abstract successor state, our process would become part of the abstract process consisting of all processes being at location b. The example shows that in two states that each have an instance of an abstract process like $(0, 1)$ these two instances may correspond to different collections of concrete processes. This is depicted in Figure 3. The problem is caused by canonical abstraction and counter abstraction collapsing processes to abstract processes. By a state change, a process migrates between instances of abstract processes. Abstraction takes away process identities and thus the means to track evolution of processes across transitions.

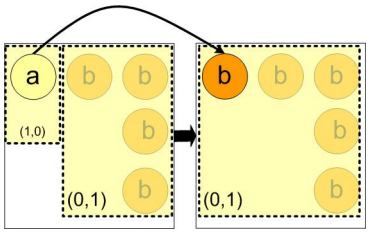


Fig. 3. Process migrating between abstract processes

1.3 Abstraction for First-Order Properties

One solution to the migration problem is to reduce the first-order model checking problem to an equivalent problem in which explicit tracking of process evolution is not necessary anymore. The semantics of universal quantification is usually given inductively in terms of the semantics of the subformula without the outermost quantifier. One combines the results of evaluating that subformula under all the different possible values the quantification variable can take on. In the domain of parameterized systems, that leaves us with an infinite number of cases to check. By applying abstraction, the infinite number of cases can be reduced to a finite, tractable number of cases.

Each subproblem only requires one to show the property for a distinguished process rater than for all processes. The abstraction can be adapted such that it is centered around the distinguished process, in that it retains more information pertaining to the distinguished process and abstracts the other processes more coarsely, and further precisely models the relation of the other processes to the distinguished process.

Note that properties which involve multiple quantifiers, like mutual exclusion, can be shown in the same way. Then there is a number of distinguished processes rather than just a single distinguished process.

Variations of this idea of decomposition are present in data type reduction, and in the shape analysis for JDBC in Ramalingam et al. [YR04].

Data type reduction Data type reduction relies on a separation of processes into two classes : a fixed number of distinguished processes and all other processes, let us call the other processes environment processes. Data type reduction retains the distinguished processes and abstracts all environment processes into one summary abstract process. The summary abstract process mimics the behavior of all the processes it represents, it is non-deterministic and memoryless, i.e. the analysis does not compute information concerning environment processes.

Figure 4 shows the data type reduction of the state from the running example. The reference process is colored black. It retains its local state a. All other processes, the environment processes, are abstracted to one abstract summary process. The local state of the summary process is abstracted away as indicated by the question tag.

1.4 Results

[SRW02] characterizes canonical abstraction in the framework of three-valued logic analysis underlying shape analysis. Abstract states are compared by a partial order, named embedding. A state being embedded in another state implies that information derived from the state that is larger in the order also holds for the smaller state. A state is always embedded in its canonical abstraction. Canonical abstraction is an abstraction which retains the optimal amount of information in the abstract. Formally, it is a tight embedding. Data type reduction is coarser. A state can be embedded into its data type reduction, however, all information about the environment process is lost, and therefore it is not a tight embedding.

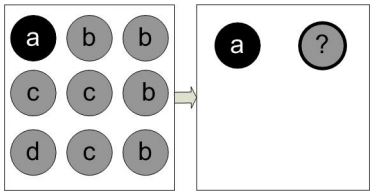


Fig. 4. Data type reduction

A more detailed treatment of the topic can be found in [Wac05] which is also available in the proceedings and on my website

`http://rw4.cs.uni-sb.de/~bwachter/thesis.pdf`

2 Related Work

Originally, canonical abstraction was designed as an abstraction technique to infer invariants of heap-manipulating programs by a technique called Three-valued Logical Analysis [SRW02] , vulgo *shape analysis*. The innovation of canonical abstraction for shape analysis was the generic *summarization of objects*, where objects were originally thought of as heap cells, and means to compute precise points-to information between abstract heap cells. This precision allows it to automatically prove partial correctness of heap-manipulating programs.

In [MYRS05], a comparison of canonical abstraction and predicate abstraction in the domain of list-manipulating programs is given. They pointed out that in principle every finitary abstraction can be expressed with predicate abstraction. However, the number of predicates needed for the encoding can be prohibitively high so that specialized abstractions can be better.

Yahav discovered [Yah01] that the algorithms from shape analysis can be generalized to parameterized protocols and Java programs. First, an abstract finite-state transition system is produced that simulates the (infinite) transition system induced by the original system. Then LTL properties are checked on the obtained transition system.

The obtained transition systems could be used to infer invariants, such as mutual exclusion, however they did not allow checking first-order *temporal* properties, as it suffers from the Migration Problem described in Section 1.2. In a subsequent paper, Yahav gave a more powerful method that is able to check properties formulated in a richer logic, termed ETL [YRSW03]. The idea was to explicitly store the evolution of processes in state transitions.

For the sake of higher efficiency and precision, later work aimed at adapting the abstraction to the particular first-order property to be checked. Ramalingam et al. describe a framework for typestate checking for Java programs [YR04], i.e. a method for checking invariants. In the context of concurrent systems, [Wac05] gave a more general model checking framework for first-order temporal properties of concurrent systems based on canonical abstraction and decomposition.

Acknowledgments

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

References

[CTV06] Edmund Clarke, Muralidhar Talupur, and Helmut Veith. Environment Abstraction for Parameterized Verification. In *VMCAI*, pages 126–141, 2006.

[DW03] Werner Damm and Bernd Westphal. Live and Let Die: LSC-based verification of UML-models. In *Formal Methods for Components and Objects, FMCO 2002*, volume 2852 of *Lecture Notes in Computer Science*, pages 99–135. Springer, 2003.

[McM00] Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.

[MYRS05] Roman Manevich, Eran Yahav, G. Ramalingam, and Mooly Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In Radhia Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*, Lecture Notes in Computer Science. Springer, jan 2005.

[PXZ] Amir Pnueli, Jessie Xu, and Lenore Zuck. Liveness with (0, 1, ?)-counter abstraction.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.

[Wac05] Björn Wachter. Checking universally quantified temporal properties with three-valued analysis. Master's thesis, Universität des Saarlandes, March 2005.

[Yah01] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3):27–40, March 2001.

[YR04] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 25–34. ACM Press, 2004.

[YRSW03] E. Yahav, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying Temporal Heap Properties Specified via Evolution Logic. In *European Symposium on Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 204 – 222. Springer-Verlag, 2003.

Formal Validation of Pattern Matching Code

Claude Kirchner, Pierre-Etienne Moreau, Antoine Reilles
INRIA & LORIA, INRIA & LORIA CNRS & LORIA
Nancy, France
First.Last@loria.fr

July 27, 2006

Abstract

When addressing the formal validation of generated software, two main alternatives consist either to prove the correctness of compilers or to directly validate the generated code. Here, we focus on directly proving the correctness of *compiled* code issued from powerful pattern matching constructions typical of ML like languages or rewrite based languages such as ELAN, Maude or Tom. In this context, our first contribution is to define a general framework for anchoring algebraic pattern-matching capabilities in existing languages like C, Java or ML. Then, using a just enough powerful intermediate language, we formalize the behavior of compiled code and define the correctness of compiled code with respect to pattern-matching behavior. This allows us to prove the equivalence of compiled code correctness with a generic first-order proposition whose proof could be achieved via a proof assistant or an automated theorem prover. We then extend these results to the multi-match situation characteristic of the ML like languages. The whole approach has been implemented on top of the Tom compiler and used to validate the syntactic matching code of the Tom compiler itself.

1 Introduction

Even if we know, since the beginning of the computer science era, that proving program correctness is profoundly difficult, the quest of software security and dependability due to the general digitalization of most human activities and process control makes this goal both inescapable and extremely important to reach.

When we deal with the previous problem, we should address the whole software conception process that we can reduce, quite schematically, to the following steps: (i) get an informal specification of the software functionalities, (ii) get a formal description of the algorithms assumed to fulfill the informal specification, (iii) get a high-level program implementation of these algorithms, (iv) get a low level program implementation of these programs, (v) get a model of the running hardware.

In this work, we restrict our interest to step (iv) and to high-level languages pattern-matching features. Therefore we address the specific problem of proving the correctness of *compiled* code issued from pattern-matching constructions appearing in high-level programming languages.

Verifiable —compiler versus compiled— code The question of compiler correctness, that is to preserve the properties of the input like its semantics and meta-properties of the underlying algorithm as its termination or the respect of heap invariants, is as old as the first compiler implementations. This is a very challenging goal since it consists in proving that *any* valid input will be correctly compiled. Furthermore, this proof has to be done every time the implementation of the compiler is modified and moreover, the compiler has itself to be compiled.

Much efforts have been done on proving correctness of parts and sometimes even complete compilers either manually [**?**, **?**, **?**, **?**] or with the help of a proof assistant. But, it is still today mostly out of reach

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/697

to prove that a program has been correctly compiled. In practice, programmers (and therefore applications) totally rely on the compilers: until one runs the program, we have no idea if the compiler has compiled the program correctly. Even extensively testing the program of course offers no guarantees. So, currently the programmer very often must blindly trust the compiler. But, most of largely used C and Fortran compilers very infrequently generate incorrect code: they are some of the most reliable software tools available. This is due to the large number of developers working to make these compilers correct, as well as the very large number of users who use these compiler, and thus contribute to their debugging. But, when designing a compiler for a new high-level language, the situation is less comfortable: on one side the number of users and written applications is small, on the other side, the introduction of new high level constructs put a lot on the compiler, and so make it even more complex to write. Since the consequences of an incorrect compiler are disastrous (all compiled programs are potentially faulty), this situation contributes to make users less confident in new languages and compiler implementations.

In this paper, we are concerned by a quite different approach consisting in proving *automatically* the correctness of the compiled code. This "skeptical" approach of the code issued from a compiler allows to deal with two kind of mis-behavior: one is due to the classical presence of an unintentional bug of the compiler. The second one concern intended hidden-behavior that could be introduced with malicious intention.

Therefore, assuming a high-level program given as input, we are considering the compiler as a black box escaping our control and we are searching to prove that the generated code is, on its own, correct. This is typical of the seminal work of [**?**] and more recently of the so called translation validation [**?**, **?**]. A comparable approach presented in [**?**] is called credible compilation and able to handle pointers in the source program. Note that this is different from the so called proof-carrying code method [**?**, **?**] which is not intended to prove the compiled program to be correct with respect to the source code, but rather on proving certain properties on the output program, such as type safety, memory safety, or the respect of a certain safety invariant.

Matching power Rewriting and pattern-matching are of general use in mathematics and informatics to describe computation as well as deduction.They are central in systems making the notion of rule an explicit and first class object, like expert and business systems (JRule), programming languages based on equational logic (OBJ) or the rewriting calculus (ELAN) or logic (Maude), functional, possibly logic, programming (ML, Haskell, Curry, Teyjus) and model checkers (Murphi). They are also recognized as crucial components of proof assistants (Coq, Isabelle) and theorem provers for expressing computation as well as strategies.

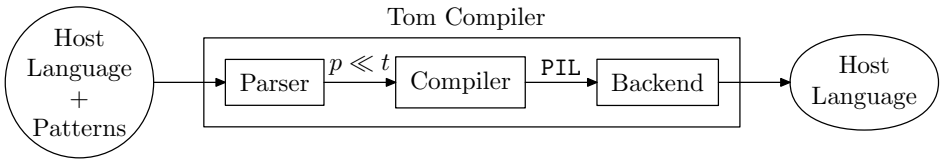
Since pattern-matching is directly related to the structure of objects and therefore is a very natural programming language feature, it is a first class citizen of functional languages like ML or Haskell and has been considered recently as a useful add-on facility in object programming languages [**?**]. This is formally backed up by works like [**?**] and particularly well-suited when describing various transformations of structured entities like, for example, trees/terms, hierarchized objects, and XML documents.

In this context, we are developing the Tom system [**?**] which provides a generic way to integrate matching power in *existing* programming languages like Java, C or ML. For example, when using Java as the host language, the sum of two integers can be described in Tom as follows:

```
Term plus(Term t1, Term t2) {
  %match(Nat t1, Nat t2) {
    x,zero   -> { return x; }
    x,suc(y) -> { return suc(plus(x,y)); }
  }
}
```

In this example, given two terms t_1 and t_2 that represent Peano integers, the evaluation of `plus` computes their sum. This is implemented by pattern-matching: t_1 is matched by the variable x, t_2 is possibly matched by one of the two patterns $zero$ or $suc(y)$. When $zero$ matches t_2, the result of the addition is x (where x has been instantiated into t_1 via matching). When $suc(y)$ matches t_2, this means that t_2 is rooted by a suc symbol: the subterm y is added to x and the successor of this number is returned. This definition of `plus` is given in a functional style, but now the `plus` function can be used elsewhere in a Java program to perform addition.

The general architecture of Tom, depicted as follows,



enlightens that a generic matching problem $p \ll t$ is compiled into an intermediate language code (`PIL`) which we would like to compute a substitution σ iff $\sigma(p) = t$. As explained in [**?**], implementing a language (possibly domain-specific) as an extension of an existing *host* language has several advantages. First, we benefit of the existing functionalities and we do not have to re-implement common language constructs. Second, the extensions themselves only need to be transformed to the point where they are expressible in the host language. Third, existing infrastructure can be reused. All these factors result into lower implementation costs and decrease the risk of building an incorrect compiler.

So, in this work we focus on proving the correctness of compiled code issued from pattern-matching constructions, and to the best of our knowledge, this is the first attempt to do so. Other works about pattern-matching compilation address in particular data abstraction e.g. [**?**], or optimizations for run-time efficiency or code size e.g. [**?**, **?**].

Roadmap of the paper When considering the notion of pattern-matching, we consider a *term* data-structure against which some *patterns* are matched. Since the host language is not fixed and could be typically either C, Java or ML, the data-model is unknown. We therefore introduce in Section 2, the notion of *formal anchor* which formally describes the relationship between the host language data-model and the algebraic notion of term and pattern.

In our language, the host language is also generic, so we have to consider an abstraction which describes the minimal set of functionality the host language should have to express the compilation of pattern-matching. This abstraction is called the *intermediate language* (`PIL`) and we define its syntax and its big-step semantics in Section 3.

Then Section 4 uses the proposed framework to define the correctness of a single pattern compilation and to show how this correctness can be reduced to the validation of a first-order proposition.

This result is then extended in Section 5 to support Caml or Tom multi-match constructs, and before concluding, Section 7 provides details about the implementation of these concepts in the current version of Tom.

2 Formal anchor

When considering the problem of proving that the behavior of a program is compatible with its semantics, we have to consider two kinds of entities. On the one side, we consider algebraic constructions, such as ground terms ($t \in \mathcal{T}(\mathcal{F})$), patterns ($p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$), and matching problems ($p \ll t$, with $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $t \in \mathcal{T}(\mathcal{F})$). On the other side, we consider programs, expressed in the `PIL` intermediate language, which are supposed to solve matching problems. We also consider data which are supposed to represent a term or a pattern. In this section, we define the notions of *representation* and *formal anchor* which define the link between algebraic entities and considered data.

2.1 Preliminary concepts

We assume the reader to be familiar with the basic definitions of first order term given, in particular, in [**?**]. We briefly recall or introduce notation for a few concepts that will be used along this paper.

A signature $\mathcal{F}$ is a set of function symbols, each one associated to a natural number by the arity function ($\mathrm{ar} : \mathcal{F} \to \mathbb{N}$). $\mathcal{F}_n$ is the subset of function symbols having n for arity, $\mathcal{F}_n = \{f \in \mathcal{F} \mid \mathrm{ar}(f) = n\}$.

$\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of *terms* built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables. A term t is said to be *linear* if no variable occurs more than once in t. Positions in a term are represented as sequences of integers and denoted by Greek letters ϵ, ν. The empty sequence ϵ denotes the position associated to the root, and it is called the root (or top) position. The subterm of t at position ν is denoted $t_{|\nu}$. $\mathcal{S}ymb(t)$ is a partial function from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ to $\mathcal{F}$, which associates to each term t its root symbol $f \in \mathcal{F}$.

The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term* and $\mathcal{T}(\mathcal{F})$ is the set of ground terms.

Two ground terms t and u of $\mathcal{T}(\mathcal{F})$ are equal, and we note $t = u$, when, for some function symbol f, $\mathcal{S}ymb(t) = \mathcal{S}ymb(u) = f$, $f \in \mathcal{F}_n$, $t = f(t_1, \ldots, t_n)$, $u = f(u_1, \ldots, u_n)$, and $\forall i \in [1..n]$, $t_i = u_i$.

A *substitution* σ is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F})$, written, when its domain is finite, $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. It uniquely extends to an endomorphism σ' of $\mathcal{T}(\mathcal{F}, \mathcal{X})$: $\sigma'(x) = \sigma(x)$ for each variable $x \in \mathcal{X}$, $\sigma'(f(t_1, \ldots, t_n)) = f(\sigma'(t_1), \ldots, \sigma'(t_n))$ for each function symbol $f \in \mathcal{F}_n$.

Given a pattern $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and a ground term $t \in \mathcal{T}(\mathcal{F})$, p *matches* t, written $p \ll t$, if and only if there exists a substitution σ such that $\sigma(p) = t$. Its negation is written $p \not\ll t$.

2.2 Object representation

Definition 1 *Given a tuple composed of a signature $\mathcal{F}$, a set of variables $\mathcal{X}$, booleans $\mathbb{B}$ and integers $\mathbb{N}$, given sets $\Omega_\mathcal{F}$, $\Omega_\mathcal{X}$, $\Omega_\mathcal{T}$, $\Omega_\mathbb{B}$, and $\Omega_\mathbb{N}$, we consider a family of* representation *functions $\ulcorner\urcorner$ that map:*

- *function symbols $f \in \mathcal{F}$ to elements of $\Omega_\mathcal{F}$, denoted $\ulcorner f \urcorner$,*
- *variables $v \in \mathcal{X}$ to elements of $\Omega_\mathcal{X}$, denoted $\ulcorner v \urcorner$,*
- *ground terms $t \in \mathcal{T}(\mathcal{F})$ to elements of $\Omega_\mathcal{T}$, denoted $\ulcorner t \urcorner$,*
- *booleans $b \in \mathbb{B} = \{\top, \bot\}$ to elements of $\Omega_\mathbb{B}$, denoted $\ulcorner b \urcorner$,*
- *natural numbers $n \in \mathbb{N}$ to elements of $\Omega_\mathbb{N}$, denoted $\ulcorner n \urcorner$.*

In other words, the representation function $\ulcorner\urcorner$ maps algebraic entities (from $\mathcal{F}$, $\mathcal{X}$, $\mathcal{T}(\mathcal{F})$, $\mathbb{B}$, and $\mathbb{N}$) to objects manipulable by the intermediate language `PIL` (elements of $\Omega_\mathcal{F}$, $\Omega_\mathcal{X}$, $\Omega_\mathcal{T}$, $\Omega_\mathbb{B}$, and $\Omega_\mathbb{N}$). We note $\ulcorner\mathcal{T}(\mathcal{F})\urcorner$ the set containing the representations of terms: $\ulcorner\mathcal{T}(\mathcal{F})\urcorner = \{\ulcorner t \urcorner | t \in \mathcal{T}(\mathcal{F})\}$, and we therefore have $\ulcorner\mathcal{T}(\mathcal{F})\urcorner \subseteq \Omega_\mathcal{T}$.

Example 1 *Let us consider $\mathcal{F} = \{e, s\}$ (with $ar(e) = 0$ and $ar(s) = 1$), and the function $\ulcorner\urcorner$ such that $\ulcorner e \urcorner = 0 \in \Omega_\mathcal{F}$, $\ulcorner s \urcorner = 1 \in \Omega_\mathcal{F}$. $\ulcorner\urcorner$ maps the symbols e and s respectively to "machine integers" 0 and 1 (i.e. the notion of integer in the intermediate language), where we assume an infinite memory. Similarly, $\ulcorner\urcorner$ can be extended to map the constant $e \in \mathcal{T}(\mathcal{F})$ to 0 ($\ulcorner e \urcorner = 0 \in \Omega_\mathcal{T}$), and any term of the form $s(x)$ to the result of the addition of 1 and the representation of x ($\ulcorner s(x) \urcorner = 1 + \ulcorner x \urcorner \in \Omega_\mathcal{T}$).*

This representation is a way to map Peano integers to "machine integers". Another well-known representation is the encoding of algebraic terms into e.g. n-ary trees.

2.3 Object mapping

In Definition 1, the notion of representation mapping has been introduced to establish a correspondence between algebraic objects and their representation in the intermediate language. However, we did not put any constraint on the representation of objects. In particular, the function $\ulcorner\urcorner$ does not necessarily preserve structural properties of algebraic objects (all terms could for example be represented by a unique constant).

Definition 2 *Given a tuple* $\langle \mathcal{F}, \mathcal{X}, \mathcal{T}(\mathcal{F}), \mathbb{B}, \mathbb{N} \rangle$*, a representation function* $\ulcorner\urcorner$*, and the mappings* $\texttt{eq} : \Omega_{\mathcal{T}} \times \Omega_{\mathcal{T}} \to \Omega_{\mathbb{B}}$*,* $\texttt{is\_fsym} : \Omega_{\mathcal{T}} \times \Omega_{\mathcal{F}} \to \Omega_{\mathbb{B}}$*, and* $\texttt{subterm}_f : \Omega_{\mathcal{T}} \times \Omega_{\mathbb{N}} \to \Omega_{\mathcal{T}}$ *($f \in \mathcal{F}$). A* formal anchor *is a mapping* $\lceil\ \rceil : \mathcal{T}(\mathcal{F}) \to \ulcorner \mathcal{T}(\mathcal{F}) \urcorner$ *such that the structural properties of* $\mathcal{T}(\mathcal{F})$ *are preserved, in* $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner$*, by the semantics of* eq*,* is_fsym*, and* $\texttt{subterm}_f$*.*
$\forall t, t_1, t_2 \in \mathcal{T}(\mathcal{F}), \forall f \in \mathcal{F}, \forall i \in [1..ar(f)]$ *we have:*

$$\begin{aligned} \texttt{eq}(\lceil t_1 \rceil, \lceil t_2 \rceil) &\equiv \lceil t_1 = t_2 \rceil \\ \texttt{is\_fsym}(\lceil t \rceil, \lceil f \rceil) &\equiv \lceil \mathcal{S}ymb(t) = f \rceil \\ \texttt{subterm}_f(\lceil t \rceil, \lceil i \rceil) &\equiv \lceil t_{|i} \rceil \text{ if } \mathcal{S}ymb(t) = f \end{aligned}$$

In the following, we always consider that the representation function is also a formal anchor. Therefore, from now on, the notation $\ulcorner\urcorner$ denotes representations that are also formal anchors.

Example 2 *In C or Java like, the notion of* term *can be implemented by a record* `(sym:integer, sub:array of term)`*, where the first slot (*`sym`*) denotes the top symbol, and the second slot (*`sub`*) corresponds to the subterms. It is easy to check that the following definitions of* eq*,* is_fsym*, and* $\texttt{subterm}_f$ *(where* $=$ *denotes an atomic equality) provide a* formal anchor *for* $\mathcal{T}(\mathcal{F})$*:*

$$\begin{aligned} \texttt{eq}(t_1, t_2) &\triangleq t_1.sym = t_2.sym \wedge \forall i \in [1..ar(t_1.sym)], \\ &\quad\ \texttt{eq}(t_1.sub[i], t_2.sub[i]) \\ \texttt{is\_fsym}(t, f) &\triangleq t.sym = f \\ \texttt{subterm}_f(t, i) &\triangleq t.sub[i] \text{ if } t.sym = f \text{ and } i \in [1..ar(f)] \end{aligned}$$

Defining a correct formal anchor is a key point to allow for the formal verification of the pattern matching code. But since this can be quite technical, we use in practice an external tool which generates for us the mapping for a given signature, as described in Section 7.

3 Intermediate language

We now describe the syntax of PIL, introduce the notion of environment, and give a formal big-step semantics ($\mapsto_{bs}$) to PIL. Informally, this intermediate language is a subset of $\mathsf{C} \cap \mathsf{Java} \cap \mathsf{ML}$ that is expressive enough to describe pattern matching procedures. This language is very close to the host language fragment it will be translated into at the end of the compilation process, and involves only a renaming of the syntactic constructions, so that proving this part of the compilation process should not present difficulties.

3.1 Syntax

Given $\mathcal{F}$, $\mathcal{X}$, $\mathcal{T}(\mathcal{F})$, $\mathbb{B}$, $\mathbb{N}$, eq, is_fsym, subterm, and a formal anchor $\ulcorner\urcorner$ as defined above, the syntax of the intermediate language PIL is defined in Figure 1.

The set of terms $\langle term \rangle$ is built over the representation of $\mathcal{T}(\mathcal{F})$, and the construct $\texttt{subterm}_f$ which retrieves the i^{th} child of a given term. The set of expressions $\langle bexpr \rangle$ contains the representation of booleans, as well as two predicates: eq which compares two terms, and is_fsym which checks that a given term is rooted by a particular symbol given in argument. The set of instructions $\langle instr \rangle$ contains only 4 instructions:

$$
\begin{array}{lll}
\texttt{PIL} & ::= & \langle instr \rangle \\
\textbf{symbol} & ::= & \ulcorner f \urcorner \ (f \in \mathcal{F}) \\
\textbf{variable} & ::= & \ulcorner x \urcorner \ (x \in \mathcal{X}) \\
\langle term \rangle & ::= & \ulcorner t \urcorner \ (t \in \mathcal{T}(\mathcal{F})) \\
 & | & \textbf{variable} \\
 & | & \texttt{subterm}_f(\langle term \rangle, \ulcorner n \urcorner) \ (f \in \mathcal{F}, n \in \mathbb{N}) \\
\langle bexpr \rangle & ::= & \ulcorner b \urcorner \ (b \in \mathbb{B}) \\
 & | & \texttt{eq}(\langle term \rangle, \langle term \rangle) \\
 & | & \texttt{is\_fsym}(\langle term \rangle, \textbf{symbol}) \\
\langle instr \rangle & ::= & \texttt{let}(\textbf{variable}, \langle term \rangle, \langle instr \rangle) \\
 & | & \texttt{if}(\langle bexpr \rangle, \langle instr \rangle, \langle instr \rangle) \\
 & | & \texttt{accept} \\
 & | & \texttt{refuse}
\end{array}
$$

Figure 1: Syntax of the intermediate language PIL

`let` and `if` correspond respectively to the assignment and the if-then-else test. We consider here that it is forbidden to assign a same variable twice. We use an *if then else* construct instead of the *switch* construct usually used to compile pattern matching because we want the generated matching algorithm to be independent of the mapping and the way terms are effectively represented. The `is_fsym` expression allows to "query" a term without the need to have function symbols as objects directly manipulated by the host language, providing more abstraction. `accept` and `refuse` are two special instructions aimed to approximate the body part of a function defined by pattern matching. In this work, since we focus on pattern matching, we only need two instructions to put the execution in a given state (`accept` or `refuse`), which denotes whether the pattern matches the subject or not.

Such a program may contain some free variables (variables which are not bound in the program by a `let` construct). They represent the input of the program, in our case the terms the pattern matching algorithm will try to match against. We call such variable *input variable*.

Assumption A. In the following, we consider that a program is evaluated in an environment where all its free variables are instantiated by a value, i.e. a term representation.

Example 3 *Given a signature $\mathcal{F} = \{a, f\}$ and a set of variables $\mathcal{X} = \{s, x\}$, a possible compilation of $f(x) \ll s$ is:*

$$
\begin{array}{l}
\texttt{if}(\texttt{is\_fsym}(\ulcorner s \urcorner, \ulcorner f \urcorner), \\
\quad \texttt{let}(\ulcorner x \urcorner, \texttt{subterm}_f(\ulcorner s \urcorner, \ulcorner 1 \urcorner), \texttt{accept}), \\
\quad \texttt{refuse} \\
)
\end{array}
$$

This program is evaluated in an environment which assigns a term representation to the free variable $\ulcorner s \urcorner$. This program checks that the root symbol of s corresponds to the representation of f. When it is the case, the first subterm of s is assigned to a variable x, and the program goes into the `accept` *state. Otherwise, it goes into the* `refuse` *state.*

On this example, it is easy to convince ourselves that the program goes into the `accept` *state if and only if the pattern effectively matches the subject. Our goal here consists to get a formal proof of this property.*

Notation. For sake of correctness, mathematical objects ($\mathbb{B}, \mathbb{N}$, and $\mathcal{X}$) have to be distinguished from their representation. However, since most of programming languages support the notion of boolean, integer and variable, when there is no ambiguity, we note $\ulcorner \top \urcorner = \texttt{true}$, $\ulcorner \bot \urcorner = \texttt{false}$, $\ulcorner 0 \urcorner = 0$, $\ulcorner 1 \urcorner = 1, \ldots, \ulcorner n \urcorner = n$ for $n \in \mathbb{N}$, and $\ulcorner x \urcorner = x$ for $x \in \mathcal{X}$.

Among the set of programs of PIL, we consider the subset of programs whose evaluation (under assumption A) always terminates in `accept` or `refuse`, whatever the input is. Those programs are called *well-formed* programs.

$$\frac{}{\Gamma, \Delta \vdash \ulcorner b \urcorner : wf \ (b \in \mathbb{B})} \qquad \frac{\Gamma, \Delta \vdash \mathtt{t}_1 : wf \quad \Gamma, \Delta \vdash \mathtt{t}_2 : wf}{\Gamma, \Delta \vdash \mathtt{eq}(\mathtt{t}_1, \mathtt{t}_2) : wf}$$

$$\frac{}{\Gamma, \Delta \vdash \ulcorner t \urcorner : wf \ (t \in \mathcal{T}(\mathcal{F}))} \qquad \frac{\Gamma, \Delta \vdash \mathtt{t} : wf}{\Gamma, \Delta \vdash \mathtt{subterm}_f(\mathtt{t}, i) : wf} \text{ if } (\mathtt{t}, f) \in \Delta \text{ and } i \in [1..\mathrm{ar}(f)]$$

$$\frac{}{\Gamma, \Delta \vdash \mathtt{accept} : wf} \qquad \frac{\Gamma, \Delta \vdash \mathtt{t} : wf \quad \Gamma :: v, \Delta \vdash i : wf}{\Gamma, \Delta \vdash \mathtt{let}(v, \mathtt{t}, i) : wf}$$

$$\frac{}{\Gamma, \Delta \vdash \mathtt{refuse} : wf} \qquad \frac{\Gamma, \Delta \vdash \mathtt{is\_fsym}(\mathtt{t}, \ulcorner f \urcorner) : wf \quad \Gamma, \Delta :: (\mathtt{t}, \ulcorner f \urcorner) \vdash i_1 : wf \quad \Gamma, \Delta \vdash i_2 : wf}{\Gamma, \Delta \vdash \mathtt{if}(\mathtt{is\_fsym}(\mathtt{t}, \ulcorner f \urcorner), i_1, i_2) : wf}$$

$$\frac{}{\Gamma, \Delta \vdash \ulcorner x \urcorner : wf} \text{ if } x \in \Gamma \qquad \frac{\Gamma, \Delta \vdash e : wf \quad \Gamma, \Delta \vdash i_1 : wf \quad \Gamma, \Delta \vdash i_2 : wf}{\Gamma, \Delta \vdash \mathtt{if}(e, i_1, i_2) : wf} \text{ if } e \neq \mathtt{is\_fsym}(\mathtt{t}, \ulcorner f \urcorner)$$

Figure 2: Type system for checking validity

Definition 3 *A program $\pi \in$ PIL is said to be well-formed when it satisfies the following properties.*

- *Each expression* $\mathtt{subterm}_f(\mathtt{t}, n)$ *is such that* $\mathtt{t}$ *belongs to* $\langle term \rangle$, $\mathtt{is\_fsym}(\mathtt{t}, \ulcorner f \urcorner)) \equiv \mathtt{true}$ *and* $n \in [1..ar(f)]$.
  (In practice, we verify that each expression of the form $\mathtt{subterm}_f(\mathtt{t}, n)$ *belongs to the* **then** *part of an instruction* $\mathtt{if}(\mathtt{is\_fsym}(\mathtt{t}, \ulcorner f \urcorner), \ldots)$.*)*
- *Each variable appearing in a sub-expression is previously initialized by a* `let` *construct, or in the evaluation environment.*

We introduce here a simple type system for verifying that a given program is well-formed, in a particular context (modeling the evaluation environment). This context is formed by the variables which have been introduced in the evaluation environment, noted Γ, and a list of couples $(\langle term \rangle, \textbf{symbol})$, noted Δ, representing the fact that in the evaluation environments, the root symbol of a given term is known.

Property 1 *A* PIL*-program π is said well-formed in an evaluation environment if and only if we can build a derivation of $\Gamma, \Delta \vdash \pi : wf$ in the type system presented in Figure 2. Γ contains the variables initialized by the environment, and Δ stores which terms have a particular root symbol.*

Proof 1 *Let π a* PIL*-program, Γ, Δ contexts such that there is a derivation $\Gamma, \Delta \vdash \pi : wf$ in the type system of Figure 2.*

So for each variable v in π, it exists contexts Γ', Δ' such that $\Gamma', \Delta' \vdash v : wf$, and thus $v \in \Gamma'$. Since v can only be introduced in the context Γ' either by early initialisation, or by applying the typing rule for `let`*, the variable v has been initialized. Also, for each* $\mathtt{subterm}_f(\mathtt{t}, i)$ *construct in π, it exists contexts Γ', Δ' such that $\Gamma', \Delta' \vdash \mathtt{subterm}_f(\mathtt{t}, i) : wf$, and $(\mathtt{t}, \ulcorner f \urcorner) \in \Delta'$. Since $(\mathtt{t}, \ulcorner f \urcorner)$ can only be introduced in the context Δ' by applying the typing rule for* $\mathtt{if}(\mathtt{is\_fsym}(\mathtt{t}, \ulcorner f \urcorner), i_1, i_2)$ *or by a previous test, the representation $\mathtt{t}$ has been checked to have root symbol f.*

Let π a well-formed PIL*-program in an evaluation environment. If we initialize the contexts Γ and Δ with the variables already instantiated in the environment, and with which terms have a particular root symbol, we can build a derivation of $\Gamma, \Delta \vdash \pi : wf$, since the typing rules for variables and* `subterm` *contructs will apply, each variable being either instantiated in the initial environment, or introduced by a* `let` *construct before its use, and root symbol of terms and arities being checked before the use of a* `subterm` *construct, either with a test in the program or in the evaluation environment.*

In practice, when verifying that a given program is well-formed, we initialize the environments with the set of input variables of the program as Γ (corresponding to the subject against which the program matches), and an empty list of couples Δ.

Notice that the well-formedness of a PIL-program is linearly decidable, since this property can be decided by the type system in Figure 2.

The program given in Example 3 is well-formed in the environment $\Gamma = \{s\}$, $\Delta = \emptyset$, since $\texttt{subterm}_f(\ulcorner s \urcorner, \ulcorner 1 \urcorner)$ is protected by the construct $\texttt{if}(\texttt{is\_fsym}(\ulcorner s \urcorner, \ulcorner f \urcorner), \ldots)$ with $1 \in [1..\mathrm{ar}(f)]$, $\ulcorner x \urcorner$ is introduced by a `let`, and $\ulcorner s \urcorner$ is in Γ.
On the contrary, the program: $\texttt{if}(\texttt{is\_fsym}(\ulcorner s \urcorner, \ulcorner f \urcorner), \texttt{if}(\texttt{eq}(\ulcorner x \urcorner, \texttt{subterm}_g(\ulcorner s \urcorner, \ulcorner 1 \urcorner)), \texttt{accept}, \texttt{refuse}), \texttt{refuse})$ is not well-formed in the same environment for two reasons: $\ulcorner x \urcorner$ is not introduced by a `let`, and $\texttt{subterm}_g$ is not guarded by an $\texttt{if}(\texttt{is\_fsym}(\ulcorner s \urcorner, \ulcorner g \urcorner), \ldots)$.

3.2 Environments

Given a matching problem, its satisfiability is of course of interest. But in most applications it is not enough and we need to compute a witness: i.e. a substitution which assigns values to the variables of the pattern. In this section, we introduce the notion of *environment*, which models the memory of a program during its evaluation. To represent a substitution, we model an environment by a stack of assignments of concrete terms to variable names. In addition, we also define a function Φ which goes back from environments to algebraic substitutions.

Definition 4 *An atomic* environment ϵ *is an assignment from* $\ulcorner \mathcal{X} \urcorner$ *to* $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner$*, written* $[x \leftarrow \ulcorner t \urcorner]$*. The composition of environments is left-associative, and written* $[x_1 \leftarrow \ulcorner t_1 \urcorner][x_2 \leftarrow \ulcorner t_2 \urcorner] \cdots [x_k \leftarrow \ulcorner t_k \urcorner]$*. Its application is such that:*

$$\epsilon[x \leftarrow \ulcorner t \urcorner](y) = \begin{cases} \ulcorner t \urcorner & \text{if } y \equiv x \\ \epsilon(y) & \text{otherwise} \end{cases}$$

We extend the notion of environment to a morphism ϵ' *from* PIL *to* PIL*, and we note* $\mathcal{E}nv$ *the set of all environments.*

Definition 5 *Given* $\mathcal{F}$ *and* $\mathcal{X}$*, we define the mapping* Φ *from environments to substitutions, by* $\Phi(\epsilon) = \sigma$ *where:*

$$\sigma = \{x_i \mapsto t_i \mid \epsilon(\ulcorner x_i \urcorner) = \ulcorner t_i \urcorner \text{ with } x_i \in \mathcal{X} \text{ and } t_i \in \mathcal{T}(\mathcal{F})\}$$

Hence, to prove the correctness of the compiled code π_p, we want to ensure that, for a given model of evaluation *"eval"* and for each term t, the following diagram commutes:

$$\begin{array}{ccc} p \ll t & \xrightarrow{compile} & \pi_p(\ulcorner t \urcorner) \\ {\scriptstyle match}\downarrow & & \downarrow{\scriptstyle eval} \\ \sigma & \xleftarrow[abstract]{} & \epsilon \end{array}$$

We are now going to make the evaluation mechanism explicit.

3.3 Big-step semantics

We use a big step semantics *à la Kahn* [**?**] to express the behavior of the PIL evaluation mechanism. The reduction relation of this big-step semantics is expressed on couples made of an environment and an instruction, denoted $\langle \epsilon, i \rangle$. The reduction relation for the big-step semantics is:

$$\langle \epsilon, i \rangle \mapsto_{bs} \langle \epsilon', i' \rangle, \text{ with } i, i' \in \langle instr \rangle, \text{ and } \epsilon, \epsilon' \in \mathcal{E}nv$$

and the rules for the big-step semantics are presented in Figure 3. The presented semantics is quite standard, however, the reader should note that conditions are evaluated modulo a formal anchor $\ulcorner \urcorner$ and the equivalences given in Section 2.3. In the line of Example 3, if we evaluate the program in the environment where s is bound to $\ulcorner f(a) \urcorner$, the condition $[s \leftarrow \ulcorner f(a) \urcorner](\texttt{is\_fsym}(s, \ulcorner f \urcorner)) \equiv \texttt{true}$ is equivalent to the condition $\ulcorner \mathcal{S}ymb(f(a)) = f \urcorner \equiv \texttt{true}$, which in this case is true since the top symbol of $f(a)$ is f.

$$\frac{}{\langle \epsilon, \texttt{accept} \rangle \mapsto_{bs} \langle \epsilon, \texttt{accept} \rangle} \quad (accept)$$

$$\frac{}{\langle \epsilon, \texttt{refuse} \rangle \mapsto_{bs} \langle \epsilon, \texttt{refuse} \rangle} \quad (refuse)$$

$$\frac{\langle \epsilon, i_1 \rangle \mapsto_{bs} \langle \epsilon', i \rangle}{\langle \epsilon, \texttt{if}(e, i_1, i_2) \rangle \mapsto_{bs} \langle \epsilon', i \rangle} \text{ if } \epsilon(e) \equiv \texttt{true} \quad (iftrue)$$

$$\frac{\langle \epsilon, i_2 \rangle \mapsto_{bs} \langle \epsilon', i \rangle}{\langle \epsilon, \texttt{if}(e, i_1, i_2) \rangle \mapsto_{bs} \langle \epsilon', i \rangle} \text{ if } \epsilon(e) \equiv \texttt{false} \quad (iffalse)$$

$$\frac{\langle \epsilon[x \leftarrow \ulcorner t \urcorner], i_1 \rangle \mapsto_{bs} \langle \epsilon', i \rangle}{\langle \epsilon, \texttt{let}(x, u, i_1) \rangle \mapsto_{bs} \langle \epsilon', i \rangle} \text{ if } \epsilon(u) \equiv \ulcorner t \urcorner \quad (let)$$

Figure 3: Big-step semantics for PIL

4 Certified compilation

Given a PIL program π and a pattern p, we first define what means for π to be a *correct compilation* of p. Intuitively, this asserts that the execution of π_p will go into the `accept` state only if the pattern p matches t. Conversely, when p does not match t, the program shall go into the `refuse` state.

Then, we state the correctness theorem and properties that show how the presented approach can be used to formally certify that a matching problem is correctly compiled. This result will be extended in Section 5 to *match* constructs as seen in Caml or Tom, where a subject is matched against multiple patterns.

4.1 Pattern-matching compilation correctness

The big-step semantics introduced previously allows us to define now the notion of *correct* compilation π_p of a pattern $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

Definition 6 *Given a formal anchor $\ulcorner\urcorner$, a well-formed program π_p is a* sound *compilation of p when both:*

$$\forall \epsilon, \epsilon' \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}),$$
$$\langle \epsilon, \pi_p(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept} \rangle \Rightarrow \Phi(\epsilon')(p) = t \quad (sound_{OK})$$
$$\langle \epsilon, \pi_p(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse} \rangle \Rightarrow p \not\ll t \quad (sound_{KO})$$

Definition 7 *Given a formal anchor $\ulcorner\urcorner$, a well-formed program π_p is a* complete *compilation of p when both:*

$$\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}),$$
$$p \ll t \Rightarrow \quad \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_p(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept} \rangle$$
$$\wedge \Phi(\epsilon')(p) = t \quad (complete_{OK})$$
$$p \not\ll t \Rightarrow \quad \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_p(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse} \rangle \quad (complete_{KO})$$

Definition 8 *A compilation of a pattern p into a program π_p is said* correct*, when it is* sound *and* complete*.*

Informally, Definition 8 says that a program π_p is a *correct* compilation of p when its execution of π_p leads to `accept` for all subjects which are matched by p, and reciprocally. The execution should also lead to `refuse` if and only if p does not match the subject. In addition, Definition 6 and 7 ensure that the environment ϵ' computed by the execution of π_p corresponds to a substitution σ such that $\sigma(p) = t$.

In order to certify that a given program π_p corresponds to a correct compilation of a pattern p, Theorem 1 shows that it is sufficient to compute all derivations of π_p to know whether the compilation is correct or not.

Theorem 1 *Given a formal anchor $\ulcorner\urcorner$, a pattern $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, and a well-formed program $\pi_p \in$ PIL, we have:*

$$\pi_p \text{ is a correct compilation of } p$$
$$\Longleftrightarrow$$
$$\forall \epsilon, \epsilon' \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}),$$
$$\langle \epsilon, \pi_p(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept} \rangle \Leftrightarrow \Phi(\epsilon')(p) = t$$

Proof 2 *By application of Properties 2 and 3 below.*

Property 2 *For all environments $\epsilon \in \mathcal{E}nv$, the derivation of a well-formed instruction $i \in \langle instr \rangle$ in the environment Γ, Δ leads trivially to* accept *or to* refuse*, and the reduction is unique.*

Proof 3 *We proceed by induction over the structure of the instruction i.*

We take as induction hypothesis that for all environments $\epsilon \in \mathcal{E}nv$, the derivation of a well-formed instruction $i \in \langle instr \rangle$ in the environment Γ, Δ leads to accept *or to* refuse*, and the reduction is unique.*

- *when $i =$* accept *(resp. $i =$* refuse*), only one inference rule can be applied: the axiom $(accept)$ (resp. $(refuse)$). So the derivation leads uniquely either to* accept *or* refuse*.*

- *when $i = \texttt{let}(x, u, i_1)$, only one inference rule can be applied: the (let) rule:*

$$\frac{\langle \epsilon[x \leftarrow \ulcorner t \urcorner], i_1 \rangle \mapsto_{bs} \langle \epsilon', i_2 \rangle}{\langle \epsilon, \texttt{let}(x, u, i_1) \rangle \mapsto_{bs} \langle \epsilon', i_2 \rangle} \text{ if } \epsilon(u) \equiv \ulcorner t \urcorner \quad (let)$$

  To complete the proof, we have to show that $\exists t \in \mathcal{T}(\mathcal{F})$ such that $\epsilon(u) \equiv \ulcorner t \urcorner$. We know that i is a well-formed instruction in the context Γ, Δ, so each variable occurring in u is previously initialized: $\epsilon(u)$ is ground. Since $u \in \langle term \rangle$, u is either a representation, a variable or a subterm$_f$*. If u is already a term representation, there is no problem. If u is a variable, u has been instantiated by term representation in the evaluation environment, since it is well-formed. The well-formed-ness of i ensures that each* subterm$_f$ *construct is encapsulated by an* if(is_fsym($\ulcorner \ldots \urcorner, \ulcorner f \urcorner$), ...) *and that each variable is initialized. Also, all* subterm$_f$ *expressions are $\equiv$-equivalent (see Definition 2) to a term representation: $\exists t \in \mathcal{T}(\mathcal{F})$ such that $\epsilon(u) \equiv \ulcorner t \urcorner$.*

  By induction hypothesis, we know that the derivation of i_1 leads either to accept *or* refuse *in a unique way, so the derivation of i is also unique and leads either to* accept *or* refuse*.*

- *when $i = \texttt{if}(e, i_1, i_2)$, using similar arguments, we show that each $\langle term \rangle$ occurring in e is $\equiv$-equivalent to a term representation. Since the expression e is either a boolean representation, an* is_fsym*, or an* eq *(where subterms are term representations), e is by definition $\equiv$-equivalent to the representation of a boolean. Thus, we have either $e \equiv$* true *or $e \equiv$* false*.*

  When $e \equiv$ true *(resp. $e \equiv$* false*), the only applicable rule is $(iftrue)$ (resp. $(iffalse)$), and we have:*

$$\frac{\langle \epsilon, i_1 \rangle \mapsto_{bs} \langle \epsilon', i_3 \rangle}{\langle \epsilon, \texttt{if}(e, i_1, i_2) \rangle \mapsto_{bs} \langle \epsilon', i_3 \rangle} \text{ if } \epsilon(e) \equiv \texttt{true} \quad (iftrue)$$

  by induction hypothesis the reduction of i_1 (resp. i_2) in the environment ϵ leads either to accept *or* refuse *in an unique way, so the reduction of i does the same.*

Thus, given $\epsilon \in \mathcal{E}nv$, the reduction of a well-formed *instruction i in an environment Γ, Δ leads either to* accept *or to* refuse*, and the reduction is unique.*

Property 3 *Given a formal anchor $\ulcorner\urcorner$ and a well-formed program $\pi_p \in$ PIL, we have:*

$$\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}), (sound_{OK}) \Rightarrow (complete_{KO}) and\, (complete_{OK}) \Rightarrow (sound_{KO})$$

Proof 4 *Let us suppose* $(sound_{OK})$ *and* $p \not\ll t$. *Since the derivation of* $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle$ *is unique (Property 2), we cannot have* $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}\rangle$ *without contradicting* $p \not\ll t$. *Property 2 says also that a derivation either leads to* `accept` *or* `refuse`. *Thus, we necessarily have* $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse}\rangle$, *and thus* $(sound_{OK}) \Rightarrow (complete_{KO})$.

Let us now suppose $(complete_{OK})$ *and that* $\exists \epsilon' \in \mathcal{E}nv$, $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse}\rangle$. *We have to show that* $p \not\ll t$. *If* $p \ll t$, *then by* $(complete_{OK})$ *we have* $\exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}\rangle$. *This is in contradiction with the uniqueness of the derivation of* $\langle \epsilon, \pi_p\rangle$, *so we have* $p \not\ll t$. *Hence* $(complete_{OK}) \Rightarrow (sound_{KO})$.

4.2 Interpreting the big-step semantics

Theorem 1 is the key result to prove that a program π_p is correct. However, the equivalence between $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}\rangle$ and $\Phi(\epsilon')(p) = t$ is difficult to prove since the big-step semantics has to be modeled and used in the proof. To solve this problem, we use a very simple form of abstract interpretation (because the symbolic simulation can be done without approximation) to statically derive a set of constraints characterizing the program behavior in the spirit of [**?**, **?**]. Therefore, given a program π_p, we compute a set of constraints $\mathcal{C}\pi_p$ such that, to prove a program correct, we show that for all t, "t satisfies $\mathcal{C}\pi_p$" if and only if "there exists ϵ such that $\Phi(\epsilon)(p) = t$".

In practice, this result is useful because the big-step semantics $\mapsto_{bs}$ does not appear anymore explicitly. This makes the proof smaller, and easier to handle by an automatic theorem prover.

Definition 9 *A big-step derivation leading to* `accept` *is called successful. Let* `s` *be an input variable and* π_p *be a* `PIL` *well-formed program. To each successful big-step derivation* $\mathcal{D}$ *we associate the conjunction* $\mathcal{C}_\mathcal{D}$ *of all constraints raised by the derivation.* $\mathcal{C}\pi_p(\texttt{s})$ *is defined as the disjunction of all constraints* $\mathcal{C}_\mathcal{D}$ *for all successful big-step derivations.*

In practice, we can use a dedicated tool to extract the constraints from a program. Starting from an environment ϵ containing only the input variable, it is sufficient to compute all big-step derivations leading to `accept`: $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}\rangle$. The constraints corresponds to the conditions raised by the application of a big-step rule, given Figure 3. Let us note that the number of generated constraints is linear in the size of the program. In practice, for a single pattern, the program is usually linear in the size of the pattern.

Given a term t, we note $\mathcal{C}\pi_p(t)$ the fact that t satisfies the constraint $\mathcal{C}\pi_p$. An example of such a constraint is given in Figure 5.

Property 4 *Given a formal anchor* $\ulcorner \urcorner$, *and a well-formed program* $\pi_p \in \texttt{PIL}$, *we have:*

$$\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}),$$
$$\exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}\rangle \Leftrightarrow \mathcal{C}\pi_p(t)$$

Proof 5 *It is clear that if the derivation* $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}\rangle$ *is possible, then* t *satisfies the constraint* $\mathcal{C}\pi_p$. *On the other hand, if* t *satisfies the constraint* $\mathcal{C}\pi_p$, *then a derivation leading to* `accept` *can be built.*

Theorem 2 *Given a formal anchor* $\ulcorner \urcorner$, *a pattern* $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, *and a well-formed program* $\pi_p \in \texttt{PIL}$, *we have:*

$$\pi_p \text{ is a correct compilation of } p$$
$$\Longleftrightarrow$$
$$\forall t \in \mathcal{T}(\mathcal{F}), \mathcal{C}\pi_p(t) \Leftrightarrow \exists \epsilon' \in \mathcal{E}nv, \Phi(\epsilon')(p) = t$$

This theorem can be used to prove correct the compilation of a pattern. As illustrated by Figure 4, given a pattern p, a condition over a term t written $\mathcal{C}p, \sigma$ can be extracted. In general this condition is of the form $\exists \sigma, \sigma(p) = t$, but, by using a matching algorithm, the substitution σ can be instantiated by $\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$, where $t_1, \ldots, t_k$ correspond to subterms of a subject t. When satisfied, this condition ensures that p matches t.

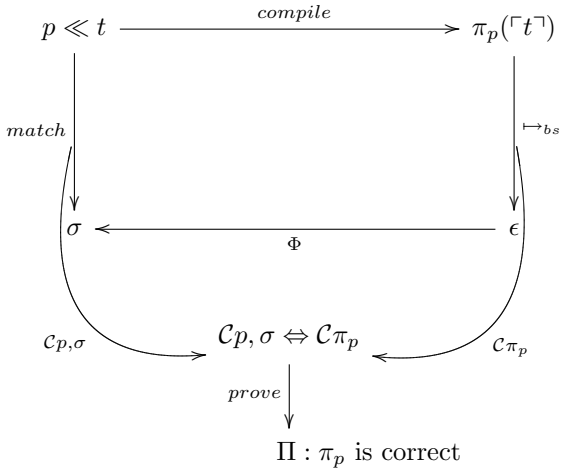


Figure 4: General schema of certification

Similarly, given a program π_p, the constraint $\mathcal{C}\pi_p$ can be computed. By application of Theorem 4 we know that if we can prove the equivalence between these two conditions, then the program π_p is a *correct* compilation of p. This proof can be done by an automatic prover to provide a formal proof Π.

4.3 Working example

As an example, let us consider the pattern $g(x, b)$, with $x \in \mathcal{X}$. Let us now suppose that our compiler produces the following program, where `s` is an input variable:

$\pi_{g(x,b)}(\mathtt{s}) \triangleq$
`if`(`is_fsym`(`s`, $\ulcorner g \urcorner$),
`let`(x_1, `subterm`$_g$(`s`, 1),
`let`(x_2, `subterm`$_g$(`s`, 2),
`let`(x, x_1,
`if`(`is_fsym`(x_2, $\ulcorner b \urcorner$), `accept`, `refuse`)))),
`refuse`)

Given a term t and an environment $\epsilon_0 = [\mathtt{s} \leftarrow \ulcorner t \urcorner]$, let us suppose that $\langle \epsilon_0, \pi_{g(x,b)}(\mathtt{s}) \rangle \mapsto_{bs} \langle \epsilon', \mathtt{accept} \rangle$. Figure 5 shows the unique derivation that can be computed by applying the inference rules defined in Section 3.3.

To make this derivation possible, the following set of constraints has to be satisfied:

$$\mathcal{C}\pi_p(\mathtt{s}) = \begin{cases} \epsilon_0(\mathtt{is\_fsym}(\mathtt{s}, \ulcorner g \urcorner)) & \equiv & \mathtt{true} & (1) \\ \epsilon_0(\mathtt{subterm}_g(\mathtt{s}, 1)) & \equiv & \ulcorner t_{|1} \urcorner & (2) \\ \epsilon_1(\mathtt{subterm}_g(\mathtt{s}, 2)) & \equiv & \ulcorner t_{|2} \urcorner & (3) \\ \epsilon_2(x_1) & \equiv & \ulcorner t_{|1} \urcorner & (4) \\ \epsilon_3(\mathtt{is\_fsym}(x_2, \ulcorner b \urcorner)) & \equiv & \mathtt{true} & (5) \end{cases}$$

(1) and (5) can be simplified using the equations of the formal anchor, (2), (3), and (4) are tautologies. Thus, to prove the correctness of π_p, we have to prove the equivalence:

$$\begin{gathered} \forall t \in \mathcal{T}(\mathcal{F}), \\ \sigma(g(x, b)) = t \wedge \sigma = \{x \mapsto t_{|1}\} \\ \Longleftrightarrow \\ \mathcal{S}ymb(t) = g \wedge \mathcal{S}ymb(t_{|2}) = b \end{gathered}$$

$$\begin{array}{l|l}
\texttt{Let}_3 \triangleq \texttt{let}(x, x_1, \texttt{if}(\texttt{is_fsym}(x_2, \ulcorner b \urcorner), \texttt{accept}, \texttt{refuse})) & \epsilon_0 = [\texttt{s} \leftarrow \ulcorner t \urcorner] \\
\texttt{Let}_2 \triangleq \texttt{let}(x_2, \texttt{subterm}_g(\texttt{s}, 2), \texttt{Let}_3) & \epsilon_1 = \epsilon_0[x_1 \leftarrow \ulcorner t_{|1} \urcorner] \\
\texttt{Let}_1 \triangleq \texttt{let}(x_1, \texttt{subterm}_g(\texttt{s}, 1), \texttt{Let}_2) & \epsilon_2 = \epsilon_1[x_2 \leftarrow \ulcorner t_{|2} \urcorner] \\
 & \epsilon_3 = \epsilon_2[x \leftarrow \ulcorner t_{|1} \urcorner]
\end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle \epsilon_3, \texttt{accept} \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}}{\langle \epsilon_3, \texttt{if}(\texttt{is\_fsym}(x_2, \ulcorner b \urcorner), \texttt{accept}, \texttt{refuse}) \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}\ \boxed{5} = \epsilon_3(\texttt{is\_fsym}(x_2, \ulcorner b \urcorner)) \equiv \texttt{true}}{\langle \epsilon_2, \texttt{let}(x, x_1, \texttt{if}(\texttt{is\_fsym}(x_2, \ulcorner b \urcorner), \texttt{accept}, \texttt{refuse})) \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}\ \boxed{4} = \epsilon_2(x_1) \equiv \ulcorner t_{|1} \urcorner}{\langle \epsilon_1, \texttt{let}(x_2, \texttt{subterm}_g(\texttt{s}, 2), \texttt{Let}_3) \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}\ \boxed{3} = \epsilon_1(\texttt{subterm}_g(\texttt{s}, 2)) \equiv \ulcorner t_{|2} \urcorner$$

$$\dfrac{\dfrac{\langle \epsilon_0, \texttt{let}(x_1, \texttt{subterm}_g(\texttt{s}, 1), \texttt{Let}_2) \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}{\langle \epsilon_0, \texttt{if}(\texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner), \texttt{Let}_1, \texttt{refuse}) \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}\ \boxed{1} = \epsilon_0(\texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner)) \equiv \texttt{true}}{\langle \epsilon_0, \texttt{if}(\texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner), \texttt{Let}_1, \texttt{refuse}) \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}\ \boxed{2} = \epsilon_0(\texttt{subterm}_g(\texttt{s}, 1)) \equiv \ulcorner t_{|1} \urcorner$$

Figure 5: Example of derivation leading to `accept`. We have $\mathcal{C}\pi_p(\texttt{s}) = \{\boxed{1} \wedge \boxed{2} \wedge \boxed{3} \wedge \boxed{4} \wedge \boxed{5}\}$

This is proved by first applying the substitution in the first part of the proof obligation, and then using the definitions of terms, symbols and subterms.

For this example, with g a function symbol of arity 2 and b a constant symbol, the mapping definition leads to the following axioms:

$$\begin{array}{l}
\forall t \in \mathcal{T}(\mathcal{F}),\ \mathcal{S}ymb(t) = g \Leftrightarrow \exists x, y \in \mathcal{T}(\mathcal{F}),\ t = g(x, y) \\
\forall t \in \mathcal{T}(\mathcal{F}),\ \mathcal{S}ymb(t) = b \Leftrightarrow (t = b) \\
\forall x, y \in \mathcal{T}(\mathcal{F}),\ g(x, y)_{|1} = x \\
\forall x, y \in \mathcal{T}(\mathcal{F}),\ g(x, y)_{|2} = y
\end{array}$$

The first two axioms define the meaning of the $\mathcal{S}ymb$ function. The two remaining axioms define the subterm function over terms rooted by the symbol g. As we use a first order prover, we need such a definition for each symbol function and for each subterm.

To prove the left to right implication, we simply apply the substitution to the left part, and then apply the first axiom, to obtain $\mathcal{S}ymb(t) = g$, and the fourth axiom to obtain $\mathcal{S}ymb(t_{|2}) = b$.

To prove the remaining implication ($\Leftarrow$), we apply the first axiom to the first constraint, obtaining $\exists x, y$ such that $t = g(x, y)$. We then apply the third and fourth axiom, to instantiate x and y by $t_{|1}$ and $t_{|2}$. The second constraint with the second axiom gives $t_{|2} = b$. We can then obtain $g(t_{|1}, b)$, and extract the substitution.

The form of such propositions is rather simple but a huge number of them could be generated, so this kind of proof should better be done by an automated theorem prover. In our implementation, we are using Zenon [**?**], a first order tableau based automatic theorem prover. One of the nice capability of Zenon is to generate a formal proof in Coq when a theorem can be proved. In our case, a witness of correctness is generated and associated to the generated code.

Another proof approach will be to use theorem proving modulo [**?**] using in particular the axioms issued from the mapping definition.

5 Extension to match constructs

Our method can be extended to support *match* constructs *à la* ML, Caml or Tom. We consider not only single patterns, but also constructs of the form `match` s `with` $(p_1 \rightarrow a_1), \ldots, (p_n \rightarrow a_n)$. The semantics of this construct is the following: if p_1 matches the subject s, the program goes into the `accept` state,

and to keep track of the pattern number, the `accept` state is labeled by p_1, noted $\texttt{accept}_{p_1}$. Otherwise, the subproblem `match` s `with` $(p_2 \to a_2), \ldots, (p_n \to a_n)$ is considered. When no pattern p_i matches s, the program goes into the `refuse` state.

This new match construct can be easily compiled using the intermediate language `PIL`. However, to avoid code duplication and to ease expression in `PIL`, it is useful to consider the *sequence* construct: $\langle instr \rangle$; $\langle instr \rangle$.

This sequence construct has the following big-step semantics:

$$\frac{\langle \epsilon, i_1 \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p \rangle}{\langle \epsilon, i_1 \ ; \ i_2 \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p \rangle} \qquad (seq_a)$$

$$\frac{\langle \epsilon, i_1 \rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse} \rangle \quad \langle \epsilon', i_2 \rangle \mapsto_{bs} \langle \epsilon'', i \rangle}{\langle \epsilon, i_1 \ ; \ i_2 \rangle \mapsto_{bs} \langle \epsilon'', i \rangle} \qquad (seq_b)$$

It is easy to show that adding the sequence rules does not break the property of uniqueness for the derivation of a well-formed instruction. The notion of correct compilation of a match construct is an extension of the definition of the correct compilation of a pattern. The difference comes from the presence of multiple patterns. Hence, when a pattern is selected to fire a rule ($\texttt{accept}_p$ in our terminology), we should ensure that all previous patterns do not match the subject. In the following, we do not make any assumptions on the form of the code to validate. This ensures that we can consider any optimizations of the matching code, like factorization of common tests.

Let $\mathcal{P}_m$ be the set of patterns for the match construct, and $<$ a total ordering relation for patterns in $\mathcal{P}_m$. In the case of ML for example, we define $<$ by the textual ordering: $p_i < p_j$ if p_i occurs before p_j in the match construct.

Definition 10 *Given a formal anchor $\ulcorner \urcorner$, a well-formed program π_m is a* sound *compilation of m when both:*

$$\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}) :$$

$$\forall p \in \mathcal{P}_m, \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p \rangle \Rightarrow \Phi(\epsilon')(p) = t \wedge (\forall p' \in \mathcal{P}_m \ s.t. \ p' < p, \Phi(\epsilon')(p') \neq t) \qquad (Msound_{OK})$$

$$\exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse} \rangle \Rightarrow \forall p \in \mathcal{P}_m, p \not\ll t \qquad (Msound_{KO})$$

Definition 11 *Given a formal anchor $\ulcorner \urcorner$, a well-formed program π_m is a* complete *compilation of m when both:*

$$\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}) :$$

$$\forall p \in \mathcal{P}_m, p \ll t \wedge (\forall p' \in \mathcal{P}_m \ s.t. \ p' < p, p' \not\ll t) \Rightarrow \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p \rangle \wedge \Phi(\epsilon')(p) = t \wedge (\forall p' \in \mathcal{P}_m \ s.t. \ p' < p, \Phi(\epsilon')(p') \neq t) \qquad (Mcomplete_{OK})$$

$$\forall p \in \mathcal{P}_m, p \not\ll t \Rightarrow \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse} \rangle \qquad (Mcomplete_{KO})$$

A compilation of a pattern p into a program π_p is said *correct*, when it is sound and complete.

Property 5 *The derivation of a well-formed instruction $i \in \langle instr \rangle$ in an environment Γ, Δ, in the extended language, leads either to $\texttt{accept}_p$ or `refuse`, and the reduction is unique.*

Proof 6 *We proceed by induction over the structure of instructions. The proof is similar to the proof of Property 2.*

We extend the type system presented Figure 2 with the rule:

$$\frac{\Gamma, \Delta \vdash i_1 : wf \quad \Gamma, \Delta \vdash i_2 : wf}{\Gamma, \Delta \vdash i_1 \; ; i_2 : wf}$$

Let $i = i_1 \; ; i_2$ be a sequence. By induction, the reduction of i_1 is unique and leads either to $\texttt{accept}_p$ or `refuse`*. In the first case, (seq_a) is applicable. The reduction of $i_1 \; ; i_2$ is equal to the reduction of i_1, so it is unique. In the second case, (seq_b) is applicable. Since the reduction of i_2 is unique, the reduction of $i = i_1 \; ; i_2$ is unique.*

Theorem 3 *Given a formal anchor $\ulcorner\urcorner$, m a match construct, and $\pi_m \in$ `PIL` a well-formed program, we have:*

$$\pi_m \textit{ is a correct compilation of } m$$
$$\Longleftrightarrow$$
$$\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}), \forall p \in \mathcal{P}_m :$$
$$\exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p\rangle$$
$$\Leftrightarrow \Phi(\epsilon')(p) = t \wedge (\forall p' \in \mathcal{P}_m \textit{ s.t. } p' < p, \Phi(\epsilon')(p') \neq t)$$

Proof 7 *We want to show, as in Property 3, that $(Msound_{OK}) \Rightarrow (Mcomplete_{KO})$ and $(Mcomplete_{OK}) \Rightarrow (Msound_{KO})$.*

In the first case, assume $(Msound_{OK})$ and $\forall p \in \mathcal{P}_m, p \not\ll t$. Since the reduction of $\langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle$ is unique, we cannot have a reduction of $\langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p\rangle$. This reduction exists, hence we have $\langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse}\rangle$. The second case can be proved in a similar way.

In order to prove that the compilation π_m of a match constructs m is correct, we have to consider each statement $\texttt{accept}_p$ in the program separately. For each pattern p in the match construct, we build all derivations in $\mapsto_{bs}$ leading to $\texttt{accept}_p$, and deduce from it a constraint, formed by a disjunction of conjunctions of single constraints. We can then for each constraint prove the corresponding proof obligation, as expressed in Theorem 3.

6 Generating the constraints

We now describe an algorithm to generate the constraints associated to a `PIL` program, and discuss the complexity of this algorithm.

6.1 Algorithm to collect constraints

The extraction starts with an environment ϵ instantiating all free variables in the program to verify, as showed in Section 4.2.

$$\begin{array}{lcl}
\mathcal{C}(\texttt{let}(x, u, i), goal) & = & \mathcal{C}(i, goal) \wedge x = u \\
\mathcal{C}(\texttt{if}(e, i_1, i_2), goal) & = & (\mathcal{C}(i_1, goal) \wedge e \equiv \texttt{true}) \\
 & & \vee \mathcal{C}(i_2, goal) \wedge e \equiv \texttt{false}) \\
\mathcal{C}(i_1 \; ; i_2, goal) & = & \mathcal{C}(i_1, goal) \vee (\mathcal{C}(i_1, \texttt{refuse}) \wedge \mathcal{C}(i_2, goal)) \\
\mathcal{C}(i, goal) & = & \top \text{ if } i = goal, \\
 & & \bot \text{ otherwise}
\end{array}$$

The algorithm computes a disjunction of conjunction of constraints. The disjuction represents the different *path* the control flow can take in the program, while the conjunctions represents the set of constraints raised in one of those path.

It is interesting to note that in the case of simple patterns, when we do not consider the ; instruction, and when there is only one occurence of `accept` in the program, then only one of such path is possible for

the control flow to reach `accept`, and so all disjuctions can be simplified by a simple boolean analysis of the generated constraint.

This function is too abstract to be used in this form, since we need for building the proof obligations of the correctness theorem the substitution built by the program when reaching `accept`. To ease the implementation we allow us to pass to the constraint extraction function the substitution built by the evaluation, and apply it to the constraints where possible.

$$\begin{array}{lcl}
\mathcal{C}(\epsilon, \texttt{let}(x,u,i), goal) & = & \mathcal{C}(\epsilon[x \leftarrow u], i, goal) \\
\mathcal{C}(\epsilon, \texttt{if}(e,i_1,i_2), goal) & = & (\mathcal{C}(\epsilon, i_1, goal) \wedge \epsilon(e) \equiv \texttt{true}) \\
 & & \vee(\mathcal{C}(\epsilon, i_2, goal) \wedge \epsilon(e) \equiv \texttt{false}) \\
\mathcal{C}(\epsilon, i_1 \,;\, i_2, goal) & = & \mathcal{C}(\epsilon, i_1, goal) \vee (\mathcal{C}(\epsilon, i_1, \texttt{refuse}) \wedge \mathcal{C}(\epsilon, i_2, goal)) \\
\mathcal{C}(\epsilon, i, goal) & = & \top \text{ if } i = goal, \bot \text{ otherwise}
\end{array}$$

In the resulting set of constraints C, we propagate variable instantiations, and apply the formal anchor equations to simplify the constraints.

In practice, this simplification is done during the extraction of the constraints, to allow detecting unsatisfiable sets of constraints (denoting an impossible path in the program flow) as early as possible, and discarding them.

6.2 Simple example

As an example, let us consider the pattern $g(x,b)$, with $x \in \mathcal{X}$. Let us now suppose that our compiler produces the following program, where `s` is an input variable:

$\pi_{g(x,b)}(\texttt{s}) \triangleq$
```
if(is_fsym(s,⌜g⌝),
   let(x1, subterm_g(s,1),
      let(x2, subterm_g(s,2),
         let(x, x1,
            if(is_fsym(x2,⌜b⌝), accept, refuse)))),
   refuse)
```

We have to start the constraint extraction with $\epsilon = [\texttt{s} \leftarrow t]$, and want to derive `accept`.

$$\begin{array}{c}
\mathcal{C}(\texttt{if}(\texttt{is_fsym}(\texttt{s},\ulcorner g \urcorner)\texttt{let}(x_1, \texttt{subterm}_g(\texttt{s},1), \\
\texttt{let}(x_2, \texttt{subterm}_g(\texttt{s},2), \texttt{let}(x, x_1, \texttt{if}(\texttt{is_fsym}(x_2,\ulcorner b \urcorner), \\
\texttt{accept}, \texttt{refuse})))), \texttt{refuse}), \texttt{accept}) \\
= \\
(\mathcal{C}(\texttt{let}(x_1, \texttt{subterm}_g(\texttt{s},1), \texttt{let}(x_2, \texttt{subterm}_g(\texttt{s},2), \\
\texttt{let}(x, x_1, \texttt{if}(\texttt{is_fsym}(x_2,\ulcorner b \urcorner), \texttt{accept}, \texttt{refuse})))), \texttt{accept}) \\
\wedge \texttt{is_fsym}(\texttt{s},\ulcorner g \urcorner) \equiv \texttt{true}) \\
\vee \\
(\mathcal{C}(\texttt{refuse}, \texttt{accept}) \wedge \texttt{is_fsym}(\texttt{s},\ulcorner g \urcorner) \equiv \texttt{false}) \\
= \\
\mathcal{C}(\texttt{let}(x_2, \texttt{subterm}_g(\texttt{s},2), \\
\texttt{let}(x, x_1, \texttt{if}(\texttt{is_fsym}(x_2,\ulcorner b \urcorner), \texttt{accept}, \texttt{refuse}))), \texttt{accept}) \\
\wedge (\texttt{is_fsym}(\texttt{s},\ulcorner g \urcorner) \equiv \texttt{true} \wedge x_1 = \texttt{subterm}_g(\texttt{s},1)) \\
\vee \\
(\bot \wedge \texttt{is_fsym}(\texttt{s},\ulcorner g \urcorner) \equiv \texttt{false})
\end{array}$$

$$
\begin{array}{c}
= \\
\mathcal{C}(\texttt{let}(x, x_1, \texttt{if}(\texttt{is\_fsym}(x_2, \ulcorner b \urcorner), \texttt{accept}, \texttt{refuse})), \texttt{accept}) \\
\wedge \texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner) \equiv \texttt{true} \wedge x_1 = \texttt{subterm}_g(\texttt{s}, 1) \wedge x_2 = \texttt{subterm}_g(\texttt{s}, 2)) \\
\vee \\
(\bot \wedge \texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner) \equiv \texttt{false}) \\
= \\
\mathcal{C}(\texttt{if}(\texttt{is\_fsym}(x_2, \ulcorner b \urcorner), \texttt{accept}, \texttt{refuse}), \texttt{accept}) \\
\wedge \texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner) \equiv \texttt{true} \wedge x_1 = \texttt{subterm}_g(\texttt{s}, 1) \\
\wedge x_2 = \texttt{subterm}_g(\texttt{s}, 2) \wedge x = x_1) \\
\vee \\
(\bot \wedge \texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner) \equiv \texttt{false}) \\
= \\
(\mathcal{C}(\texttt{accept}, \texttt{accept}) \\
\wedge \texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner) \equiv \texttt{true} \wedge x_1 = \texttt{subterm}_g(\texttt{s}, 1) \\
\wedge x_2 = \texttt{subterm}_g(\texttt{s}, 2) \wedge x = x_1 \wedge \texttt{is\_fsym}(x_2, \ulcorner b \urcorner) \equiv \texttt{true}) \\
\vee \\
(\mathcal{C}(\texttt{refuse}, \texttt{accept}) \\
\wedge \texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner) \equiv \texttt{true} \wedge x_1 = \texttt{subterm}_g(\texttt{s}, 1) \\
\wedge x_2 = \texttt{subterm}_g(\texttt{s}, 2) \wedge x = x_1 \wedge \texttt{is\_fsym}(x_2, \ulcorner b \urcorner) \equiv \texttt{false}) \\
\vee \\
(\bot \wedge \texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner) \equiv \texttt{false}) \\
= \\
(\top \wedge \texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner) \equiv \texttt{true} \wedge x_1 = \texttt{subterm}_g(\texttt{s}, 1) \wedge x_2 = \texttt{subterm}_g(\texttt{s}, 2) \\
\wedge x = x_1 \wedge \texttt{is\_fsym}(x_2, \ulcorner b \urcorner) \equiv \texttt{true}) \\
\vee \\
(\bot \wedge \texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner) \equiv \texttt{true} \wedge x_1 = \texttt{subterm}_g(\texttt{s}, 1) \\
\wedge x_2 = \texttt{subterm}_g(\texttt{s}, 2) \wedge x = x_1 \wedge \texttt{is\_fsym}(x_2, \ulcorner b \urcorner) \equiv \texttt{false}) \\
\vee \\
(\bot \wedge \texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner) \equiv \texttt{false})
\end{array}
$$

6.3 Simplifying constraints

Using the algorithm to collect constraints produces a huge formula, containing many $\top$ and $\bot$, so we can first simplify this formula as a boolean formula.

We apply the following rewrite system, with a letftmost-innermost strategy:

$$
\begin{array}{lcl}
\bot \wedge x & \rightarrow & \bot \\
x \wedge \bot & \rightarrow & \bot \\
\top \vee x & \rightarrow & \top \\
x \vee \top & \rightarrow & \top \\
\bot \vee x & \rightarrow & x \\
x \vee \bot & \rightarrow & x \\
\neg \top & \rightarrow & \bot \\
\neg \bot & \rightarrow & \top
\end{array}
$$

This rewrite system simplify the boolean constraints, to obtain the simplified contraints.

The extracted constraints for $\pi_{g(x,b)}$ are:

$$\begin{array}{c} \texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner) \equiv \texttt{true} \\ \wedge \quad x_1 = \texttt{subterm}_g(\texttt{s}, 1) \\ \wedge \quad x_2 = \texttt{subterm}_g(\texttt{s}, 2) \\ \wedge \quad x = x_1 \\ \wedge \quad \texttt{is\_fsym}(x_2, \ulcorner b \urcorner) \equiv \texttt{true} \end{array}$$

Those constraints can be simplified using the definitions of the mapping,

$$\begin{array}{rcl} \texttt{eq}(\ulcorner t_1 \urcorner, \ulcorner t_2 \urcorner) & \equiv & \ulcorner t_1 = t_2 \urcorner \\ \texttt{is\_fsym}(\ulcorner t \urcorner, \ulcorner f \urcorner) & \equiv & \ulcorner \mathcal{S}ymb(t) = f \urcorner \\ \texttt{subterm}_f(\ulcorner t \urcorner, \ulcorner i \urcorner) & \equiv & \ulcorner t_{|i} \urcorner \text{ if } \mathcal{S}ymb(t) = f \end{array}$$

The mapping equalities can be oriented, to be used as a rewrite system. Informally, the goal is to transform a constraint giving information about the objects manipulated by the program into a constraint giving information about the algebraic terms used at source level.

$$\begin{array}{rcl} \texttt{eq}(\ulcorner t_1 \urcorner, \ulcorner t_2 \urcorner) & \rightarrow & \ulcorner t_1 = t_2 \urcorner \\ \texttt{is\_fsym}(\ulcorner t \urcorner, \ulcorner f \urcorner) & \rightarrow & \ulcorner \mathcal{S}ymb(t) = f \urcorner \\ \texttt{subterm}_f(\ulcorner t \urcorner, \ulcorner i \urcorner) & \rightarrow & \ulcorner t_{|i} \urcorner \text{ if } \mathcal{S}ymb(t) = f \end{array}$$

Applying this rewrite system to the constraints generated for $\pi_{g(x,b)}$ produces the constraint:

$$\begin{array}{c} \mathcal{S}ymb(\texttt{s}) = g \\ \wedge \quad x_1 = \texttt{s}_{|1} \\ \wedge \quad x_2 = \texttt{s}_{|2} \\ \wedge \quad x = x_1 \\ \wedge \quad \mathcal{S}ymb(x_2) = b \end{array}$$

6.4 Size of the generated formula

Let us consider the size of the input as the number of nodes of the input abstract syntax tree.

All rules except the extraction rule for the sequence ; are linear, each node in the input abstract syntax tree is visited once.

The rule for sequence visits the left subtree of ; twice, thus leading to an exponential in the number of ; nodes in the input. However, the sequence ; is associative, so we can consider that the left subtree of a sequance is not itself a sequence, and the generated code usually do not contain many nested levels of sequences, so this exponential do not appear in practical use.

6.5 Decidability

The theorems we want to prove have their only quantifiers at the root, introducing the input variables.

They correspond to the second part or the correctness theorem [] and if they hold, prove that the compilation of the corresponding patterns was successful. $\mathcal{C}\pi_p$ is the constraint formula produced by the application of the constraint extraction algorithm on the PIL program π_p, corresponding to the compilation of the matching problem p. $\mathcal{C}\pi_p(t)$ represents the fact that the term t satisfy this constraint formula.

Theorem 4 *Given a formal anchor $\ulcorner \urcorner$, a pattern $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, and a well-formed program $\pi_p \in$ PIL, we have:*

$$\begin{array}{c} \pi_p \textit{ is a correct compilation of } p \\ \Longleftrightarrow \\ \forall t \in \mathcal{T}(\mathcal{F}), \mathcal{C}\pi_p(t) \Leftrightarrow \exists \epsilon' \in \mathcal{E}nv, \Phi(\epsilon')(p) = t \end{array}$$

The part we want to prove automatically is:

$$\forall t \in \mathcal{T}(\mathcal{F}), \mathcal{C}\pi_p(t) \Leftrightarrow \exists \epsilon' \in \mathcal{E}nv, \Phi(\epsilon')(p) = t$$

Since the output substitution can be computed from the extracted constraints (and directly, using the second extraction algorithm), we can also remove from the property we have to prove the existential quantifier, and use this computed substitution in place of ϵ'.

The property to prove becomes:

$$\forall t \in \mathcal{T}(\mathcal{F}), \mathcal{C}\pi_p(t) \Leftrightarrow \Phi(\epsilon)(p) = t$$

Proving that: $\forall t \in \mathcal{T}(\mathcal{F}), \Phi(\epsilon)(p) = t \Rightarrow \mathcal{C}\pi_p(t)$ can be done by orienting the axioms defining the subterm and symbol property.

7 Early Experimental Results

The presented work has been implemented and applied to the intermediate language of Tom [**?**]. Tom is a language extension which adds pattern matching primitives to C, Java, and Caml. One particularity is to provide support for matching modulo sophisticated theories, like associative operators with neutral element. However, in this work, we only considered the case of the empty theory (i.e. syntactic matching), with possibly non-linear patterns.

Tom is based on the notion of formal anchor presented in Section 2.3. Thus, it is data structure independent, and customizable for any term implementation. Considering a simple term implementation in C, for example, we can define the following anchor:

```
struct term { int symbol;
              int arity;
              struct term **subterm; };
%typeterm Term {
 implement        { struct term*      }
 get_subterm(t,n) { t->subterm[n]     }
 equal(t1,t2)     { term_equal(t1,t2) }
}
%op Term a       { is_fsym(t) { t->symbol == A } }
%op Term b       { is_fsym(t) { t->symbol == B } }
%op Term f(Term) { is_fsym(t) { t->symbol == F } }
```

Given a `%match` construct, as illustrated by Figure 6, the compiler translates patterns into `PIL` instructions, which use the previously defined formal anchor. In practice, this mapping is supposed correct, in the sense that structural properties of terms should be preserved. To simplify this task, when no particular data-structure is required, a generator of term based implementations, coupled with a generator of formal anchors, can be used [**?**].

To prove the generated `PIL` code correct, we recently added to Tom a component (constraints extractor) which generates, for each pattern p, the constraints $\mathcal{C}\pi_p$ and $\mathcal{C}p, \sigma = (\exists \sigma,\ \sigma(p) = t)$, where t is the input term. In a second step, these two constraints are sent to a prover to show their equivalence. To experiment our approach, we used Zenon, because, in addition to be fully automatic, it can generate a Coq formal proof when it succeeds. This is essential in our "skeptical" approach since it allows the user of the generated program to verify the proof by himself.

The verification tool is integrated into the Tom architecture, but note that no support from the internal compiler is needed: $\mathcal{C}p, \sigma$ are extracted from the AST produced by the parser, and $\mathcal{C}\pi_p$ are extracted from the `PIL` program produced by the compiler, or any other component such as an optimizer for example. Seeing the compiler as a black-box allows us to perform any kind of optimization unless `PIL` code is generated. At the moment we handle only the intermediate code of the compiler, ignoring the parser and the code generator.

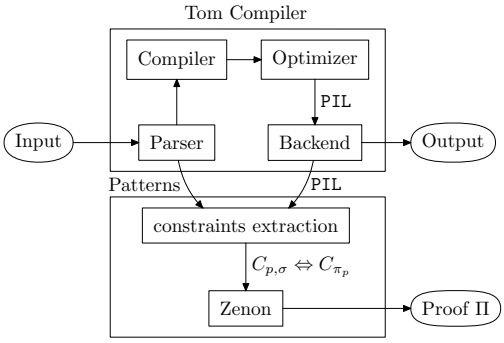


Figure 6: Global architecture of Tom

In our case, the backend performs a so straightforward *one-to-one* translation, from PIL instructions to host language instructions, that we trust in its correctness.

The interest of this approach is to allow to verify code corresponding to an optimized many-to-one algorithm, where common tests are factored.

To illustrate the applicability of the present approach, we tested our validator on several small examples, all of which worked with success, in an efficient way. For a more realistic test, to show how the approach scales to biggest problems, we generated proof obligations corresponding to the compilation of Tom itself (written in Tom). 184 patterns were extracted by the parser, after discarding associative patterns. From this set of patterns, 1018 applications of inference rules were needed to compute all derivations which lead to `accept`. This step generated 834 constraints ($\mathcal{C}\pi_p$). Most of them were tautologies of the form $\epsilon(\texttt{subterm}_g(\texttt{s}, 1)) \equiv \texttt{s}_{|1}$. After a first step of simplification, 273 remaining constraints were simplified using 2533 $\equiv$-equivalence relation steps. On a PowerMac G5 (2GHz), the compilation of Tom, with the generation of theorems to prove, only increased the compilation time by 20% (going from 70 seconds to 84 seconds). This clearly shows that the approach can scale to large applications. In the current implementation, the translation to Zenon formalism is done fully automatically starting from the Tom program.

8 Conclusion and future works

When using a compiler, we always think it is correct. When writing a compiler, we know it is incorrect. This drives us to present a framework addressing the specific problem of generated pattern matching code validation. The main benefit of such an approach over a traditional compiler is that the compiler output is formally checked after each compilation, thus simplifying testing and development and providing a way to prove the formal validity of the generated code. We have seen that the proposed approach is powerful and flexible enough to validate the compilation of *match* constructs *à la* Caml or Tom.

We are now attacking the challenging problem of extending this method to support matching with associative (list) operators, like those of the Tom language, and with associative-commutative operators, like in many rewriting based languages like ELAN. Matching modulo theories is much more elaborated than syntactic matching, and so is writing such a pattern matching compiler. Validation of the produced code can then help developing and debugging new optimizations for these matching theories. Furthermore, when matching modulo theories, a new completeness problem has to be solved: the generated matching code has not only to find a substitution if the matching problem has one, but may have to produce *all* possible solutions for the matching problem.

The approach proposed here generates proof obligations of a very strict form. These proof obligations are in general easy to prove, but we should investigate our current conjecture that this class of problems is indeed decidable.

Although we were only interested in this work in the correctness of the generated code against the source

problem, some additional properties of the source system could be proved by this method. For example the completeness of definition of a function defined by pattern matching could be proved by showing that there is no possible reduction to `refuse`.

Finally, our ultimate goal is to formally prove the correct compilation of the normalization process induced by a rewrite system. Proving the correct compilation of rewrite system execution will allow us to safely deduce on the program produced by the compilation of a rewriting specification the properties proved for this specification, like termination or confluence. This paper is a first but crucial step in this direction.

Acknowledgments: We would like to thank Luigi Liquori for his inside full reading on a preliminary version of this paper, Pierre Weis and Germain Faure for useful comments on this work. Special thanks are due to Damien Doligez for fruitful discussions and his help in connecting our tool to Zenon. We also thank the anonymous referees for valuable comments and suggestions that led to a substantial improvement of the paper.

References

[1] Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[2] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 1998. IEEE.

[3] Robert S. Boyer and Yuan Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In D. Kapur, editor, *Proceedings of the Eleventh International Conference on Automated Deduction*, pages 416–430. Springer-Verlag, 1992.

[4] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Matching Power. In Aart Middeldorp, editor, *Proceedings of RTA'2001*, volume 2051 of *LNCS*, Utrecht (The Netherlands), May 2001. Springer-Verlag.

[5] Damien Doligez. Zenon: an automatic theorem prover for first-order logic. Available as part of the Focal system at http://focal.inria.fr/download.

[6] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, Nov 2003.

[7] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 26–37. ACM Press, 2001.

[8] Sumit Gulwani and George C. Necula. Global value numbering using random interpretation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 342–352. ACM Press, 2004.

[9] Gilles Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, 1987. Springer-Verlag.

[10] Hélène Kirchner and Pierre-Etienne Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

[11] Andreas Krall and Jan Vitek. On extending java. In *Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*, pages 321–335. Springer-Verlag, 1997.

[12] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc. 29th ACM Symposium on Principles of Programming Languages*, pages 283–294. ACM Press, 2002.

[13] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Proceedings Symposium in Applied Mathematics, Vol. 19*, pages 33–41. AMS, 1967.

[14] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.

[15] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 144–152. ACM Press, 1973.

[16] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[17] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 39(4):612–625, 2004.

[18] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified PreScheme compiler. *Lisp Symb. Comput.*, 8(1-2):111–182, 1995.

[19] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166. Springer-Verlag, 1998.

[20] Martin C. Rinard and Darko Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[21] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2004.

[22] Mark. G. J. van den Brand, Pierre-Etienne Moreau, and Jurgen Vinju. Generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings - Software Engineering*, 152(2):70–79, April 2005.

[23] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 307–313. ACM Press, 1987.

[24] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *Proceedings of the 5th ACM SIGPLAN international Conference on Principles and Practice of Declarative Programming*, pages 264–274. ACM Press, 2003.

Formalizing On Chip Communications in a Functional Style

Julien Schmaltz[1] and Dominique Borrione[2]

[1] Saarland University, Department of Computer Science
FR 6.2 Informatik, PostFach 151150, 66 041 Saarbrücken,Germany
julien@cs.uni-sb.de
[2] TIMA Laboratory - VDS Group
46, avenue Felix Viallet, 38031 Grenoble Cedex, France
Dominique.Borrione@imag.fr

Abstract. This paper presents a formal model for representing *any* on-chip communication architecture. This model is described mathematically by a function, named *GeNoC*. The correctness of *GeNoC* is expressed as a theorem, which states that messages emitted on the architecture reach their expected destination without modification of their content. The model identifies the key constituents common to *all* communication architectures and their essential properties, from which the proof of the *GeNoC* theorem is deduced. Each constituent is represented by a function which has no *explicit* definition but is constrained to satisfy the essential properties. Thus, the validation of a *particular* architecture is reduced to the proof that its concrete definition satisfies the essential properties. In practice, the model has been defined in the logic of the ACL2 theorem proving system. We define a methodology that yields a systematic approach to the validation of communication architectures at a high level of abstraction. To validate our approach, we exhibit several architectures that constitute concrete instances of the generic model *GeNoC*. Some of these applications come from industrial designs, such as the AMBA AHB bus or the Octagon network from ST Microelectronics.

1 Introduction

Current chip technology (65nm) allows the integration of several hundred million transistors on a single die, which requires a huge progress in design methodologies. Indeed, chip business is highly competitive and time to market has shrunk. A three month delay induces the loss of one fourth of the expected income [6]. To face this increasing time pressure, *systems on a chip* (SoC) are designed through a *platform* based approach: a new SoC is built according to a generic architecture, using pre-designed parameterized modules and processor cores. In that context, the interconnect structure becomes challenging both for design and verification [26].

Until recently, most of the verification effort was spent on the processing elements, and the literature specifically devoted to the embedded communication

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/700

architecture is relatively sparse. Bus architectures, and their protocols, have been the subject of the earlier works on that topic. Roychoudhury *et al.* use the SMV model checker to debug an academic implementation of the AMBA AHB protocol [19]. Their model is written at the register transfer level and without any parameter. Roychoudhury *et al.* detect a live lock scenario that was caused by the implementation of their arbiter rather than by the protocol itself. More recently, Amjad [2] used a model checker, implemented in the HOL theorem prover, to verify the AMBA APB and AHB protocols, and their composition in a single system. Using model checking, safety properties are verified on each protocol individually. The HOL tool is used to verify their composition. In this work also, the model is at a low level of abstraction, and without any parameter.

Networks on a chip (NoC) are a more recent design paradigm, and little work has been done about their formal verification outside straightforward model checking on fixed structures. A notable exception is the work of Gebremichael *et al.* [10], who recently specified the Æthereal protocol [11] of Philips in the PVS logic. The main property they verified is the absence of deadlock for an arbitrary number of masters and slaves.

At this point, it is worth noting that the above mentioned formal verification efforts, devoted to communication architectures and protocols, were performed at the register transfer level (RTL), on a very specific design. This level was considered appropriate when the same source was generating the synthesizable design for the full system. With the advent of outsource IP's and platform based design, the current trend in the SoC design community is to raise the level of abstraction [26] and rely on verified parameterized library modules. This requirement will soon extend to communication network kernels, yet a formal theory for this category of functional modules is non existing today. In effect, most textbook (*e.g.* [8]) *describe* architectures in an informal manner.

On the path to the definition of a formal theory of communications, two important studies have already treated part of it. Moore [17] defined a formal model of asynchrony by a function in the Boyer-Moore logic [5], and showed how to use this general model to verify a biphase mark protocol. More recently, Herzberg and Broy [12] presented a formal model of stacked communication protocols, in the sense of the OSI reference model. In a relational framework supporting a component-oriented view, they defined operators and conditions to navigate between protocol layers. Herzberg and Broy's framework considers all OSI layers. Thus, it is more general than Moore's work, which is targeted at the lowest layer. In contrast, Moore provides mechanized support. Both studies focus on protocols and do not consider the underlying interconnection structure explicitly.

The long term objective of our research is to support the validation of abstract specifications for on chip communication architectures, and the verification of their correct implementation by a given, possibly parameterized, IP. To this aim, we provide a general formal framework that encompasses the essential constituents of communication modules - *i.e.* protocols *and* topologies, routing algorithms and scheduling policies - and applies to a wide variety of communi-

cation architectures. It is essential that our theory be directly expressible in the logic of an interactive theorem prover, either first or higher order, to provide mechanized reasoning support.

This paper presents what constitutes, to our knowledge, a first proposal for a formal theory for communication architectures. Communications on the chip share many concepts with computer networks, but work on a different time scale. Systems on a chip often have very hard time, heat and power constraints. On chip communications must be predictable: losing and resending a message, or reordering message pieces is unacceptable within a SoC, while it is current practice on the Internet. On chip communications are more constrained, and their topology is statically defined, which simplifies the protocols. This paper only deals with these restricted communications systems: buses and NoC's. We formalize a generic communication architecture in functional form. A global function, called *GeNoC*, formalizes the interactions between the three key constituents: interfaces, routing and scheduling.

This generic model makes no assumption on the protocol, the topology, the routing algorithm, or the scheduling policy. To abstract from any particular architecture, we have identified essential properties (considered proof obligations or simply constraints) for each constituent. Those imply the overall correctness of *GeNoC*. Hence, the validation of any particular architecture is reduced to the proof that each one of its constituents satisfies the generic constraints. By embedding our theory in the logic of an automated proof assistant, we provide a tool to specify and to validate network on a chip description at a high level of abstraction. For any concrete architecture, the proof assistant automatically generates the proof obligations that must be satisfied to prove the compliance of this architecture with our theory.

This paper is structured as follows. The next section presents a motivating example network, and defines our notations. Section 3 gives an overview of our theory. Section 4 constitutes the core of the paper and our original contribution: it precisely defines the functions and proof obligations for the main constituents of a network on chip. Section 5 exposes our methodology for applying our model to a practical network on chip in a systematic way, and gives an overview of our experiments on a variety of communication architectures. The instanciation of the *GeNoC* model to the "Octagon" design by STMicroelectronics is used as an illustration. Finally, section 6 concludes the paper and gives future research directions.

2 Background for the Theory

Our theory relies on some background principles and fundamental common features of all communication architectures. To make our theory easily expressible in interactive proof assistants, we define it using lists and their associated operators, as introduced at the end of this section. Let us first start with an example.

2.1 An NoC Example: the Octagon

This network on a chip has been designed by STMicroelectronics [14]. A basic Octagon unit consists of eight nodes and twelve bidirectional links (Figure 1). It has two main properties: the communication between any pair of nodes requires at most two hops, and it has a simple, shortest-path routing algorithm [14].

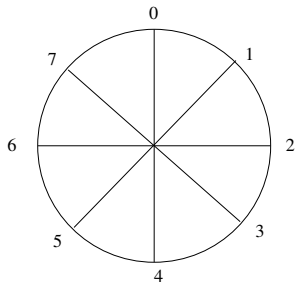


Fig. 1. Basic Octagon Unit

An *Octagon packet* is data that must be carried from the source node to the destination node as a result of a communication request by the source node. A scheduler allocates the entire path between the source and destination nodes of a communicating node pair. Non-overlapping communication paths can occur concurrently, permitting spatial reuse.

The routing of a packet is accomplished as follows. Each node compares the tag ($PackAd$) to its own address ($NodeAd$) to determine the next action. The node computes the relative address of a packet as:

$$RelAd = (PackAd - NodeAd) \bmod 8 \tag{1}$$

At each node, the route of packets is a function of $RelAd$ as follows:

- $RelAd = 0$, process at node
- $RelAd = 1$ or 2, route clockwise
- $RelAd = 6$ or 7, route counterclockwise
- route across otherwise

Example 1. Consider a packet *Pack* at node 2 sent to node 5. First, $5-2 \bmod 8 = 3$, *Pack* is routed across to 6. Then, $5 - 6 \bmod 8 = 7$, *Pack* is routed counterclockwise to 5. Finally, $5 - 5 \bmod 8 = 0$, *Pack* has reached its final destination.

2.2 A Unifying Model

The previous example is generalized to the communication model of Figure 2. An arbitrary but finite number of *nodes* is connected to some communication

architecture, bus or network. Topologies, routing algorithms and scheduling policies are its essential constituents. To distinguish between interface-application and interface-interface communications, an interface and an application communicate using *messages*; two interfaces communication using *frames*.

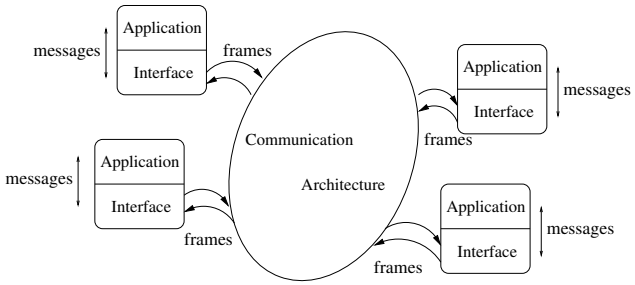


Fig. 2. Communication Model

Applications represent the computational and functional aspects of nodes. They are either active or passive. Typically, active applications are processors and passive applications are memories. We consider that each node contains one passive and one active application, *i.e.* each node is capable of sending and receiving frames. As we want a general model, applications are not considered *explicitly*: passive applications are not actually modeled, and active applications are reduced to the list of their pending communication requests. We focus on communications between distant nodes. We suppose that in every communication, the destination node is distinct from the source node.

2.3 Lists: Notations and Operators

Lists are essential to the implementation of our formalism. We briefly present the notations and the functions used to manipulate them. Notations about lists are summarized in Table 1.

Letters l or L are used to denote a list or a list of lists. List elements are often represented by letter e. The empty list is denoted by ϵ. A list l is a finite sequence of k values indexed from 0 to $k-1$, $l = (l[i])_{i\in[0;k-1]}$.

$Len(l)$ returns the length of list l (its number of elements), and $Last(l)$ returns its last element. Predicate $NoDuplicatesp(l)$ recognizes a list in which each two elements are distinct. The type of a list l_1 is defined by the membership of its elements to a given set E, and is denoted with the $\subseteq_l$ operator.

Adding an element e in front of a list l creates a new list l', noted $l' = e.l$. Element e takes index 0 in l'. Elements of l' with an index i greater that 0 are elements of l with index $i-1$. If the list is a list of lists, e is a list. The append of two lists, l_1 and l_2, of the same type is denoted $l_1 \sqcup l_2$, resulting in a list of

Name	Purpose
$e.l$	add element e to list l
$l_1 \subseteq_l E$	l_1 is a list of E (type)
$l_1 \sqcup l_2$	append of l_1 and l_2
$e \in_l l_1$	e is an element of list l_1
$l_1 \sqsubseteq l_2$	l_1 is included in l_2
$l_1 \sqcap l_2$	elements common to l_1 and l_2
$List(l_1, l_2)$	juxtaposition of lists l_1 and l_2
$Len(l)$	the number of elements contained in l
$Last(l)$	the last element of l
$NoDuplicatesp(l)$	recognizes a list l with no duplicate
ϵ	the empty list
$l[i]$	an element of list l, $0 \leq i \leq Len(l) - 1$

Table 1. Notations and functions used to manipulate lists

this type. If the lists have not the same type, their juxtaposition is obtained by function $List(l_1, l_2)$. An element e is an element of a list l if and only if e is a value of l. $e \in_l l_1$ reads: e is an element of list l_1. A list l_1 is *included* in a list l_2 , denoted $l_1 \sqsubseteq l_2$, if and only if every element of l_1 is an element of l_2. The empty list, ϵ, is included in all lists. Examples: the list $(1\ 1\ 1)$ is included in the list (1); the list $(3\ 2)$ is included in the list $(1\ 2\ 3)$. The list l in which the first occurrence of an element e has been removed is noted $l \setminus e$. The list l' containing all the elements that are elements of lists l_1 and l_2 is noted $l' = l_1 \sqcap l_2$. This list preserves the element ordering of l_1. For instance, $(1\ 2\ 5\ 3) \sqcap (1\ 2\ 1\ 3\ 4) = (1\ 2\ 3)$. The definition of operator $\sqcap$ is as follows:

$$l_1 \sqcap l_2 \triangleq \begin{cases} \epsilon & \textbf{if } l_1 = \epsilon \vee l_2 = \epsilon \\ l'_1 \sqcap l_2 & \textbf{if } l_1 = e.l'_1 \wedge e \notin_l l_2 \\ e.(l'_1 \sqcap (l_2 \setminus e)) & \textbf{if } l_1 = e.l'_1 \wedge e \in_l l_2 \end{cases} \quad (2)$$

If the elements e of a list L are lists, the list of the elements of L with the same index i in each e is noted $L_{\lfloor i}$.

In our model, the meaning of the elements of e is often given by an identifier. For readability, we shall use the identifier rather than its index. For instance, assume that e is a list composed of a key, a name and a surname: $e = (key\ name\ surname)$. Let L be a list of elements e of this kind. The list of the keys is noted $L_{\lfloor key}$, the list of the names $L_{\lfloor name}$ and the list of the surnames $L_{\lfloor surname}$.

Very often, a list is built by the application of a function f to every element of a list l. This operation corresponds to a higher-order function φ that takes as arguments a function f and a list l. Function φ returns the list of the results of the application of f to every element of l [1]. As function f could be complex, it is not always practical to have it explicitly formulated. Often, it suffices to express the modification done on each element. To alleviate the notation, the

[1] In functional programming, this corresponds to the map operation

application of function φ is noted using operator Λ defined as follows:

$$\bigwedge_{e \in l} f(e) \equiv \varphi(l, f) \triangleq \begin{cases} \epsilon & \textbf{if } l = \epsilon \\ f(e).\varphi(l', f) & \textbf{otherwise } l = e.l' \end{cases} \quad (3)$$

For instance, let l be a list of integer couples $e = (x_1\ x_2)$. The list l' of the sums $x_1 + x_2$ over the elements e of l is easily defined with operator Λ:

$$l' = \bigwedge_{e \in l} (e[0] + e[1])$$

3 Model Overview

3.1 Principles of *GeNoC*

Function *GeNoC* represents the transmission of messages on a generic communication architecture, with an arbitrary topology, routing algorithm and switching technique. Its main argument is the list of messages emitted at source nodes. It returns the list of the results received at destination nodes. Its definition mainly relies on the following functions:

1. *Interfaces* are represented by two functions: *send* encapsulates a message into a frame and injects the frame on the network; *recv* decodes the frame to recover the emitted message. The main constraint associated to these functions expresses that a receiver should be able to extract the encoded information, *i.e.* the composition of functions *recv* and *send* ($recv \circ send$) is the identity function. Note that this property is also present in Moore's model of asynchrony, as well as in Herzberg and Broy's framework.
2. *Routing and topology* are represented by function *Routing*. The routing algorithm consists of the successive application of unitary moves. For each pair made of a source s and a destination d, *Routing* computes *all* the possible routes allowed by the unitary moves. The main constraint associated to *Routing* is that each route from s to d effectively starts in s and uses only existing nodes to end in d.
3. *The switching technique* is represented by function *Scheduling*. The scheduling policy participates in the management of conflicts, and computes a set of possible simulatenous communications. Formally, these commutations satisfy an *invariant*. Scheduling a communication, *i.e.* adding it to the current set of authorized communications, must preserve the invariant, at all times and in any admissible state of the network. The invariant is specific to the scheduling policy. In our formalization, the existence of this invariant is assumed but not explicitly represented. From a list of requested communications, function *Scheduling* extracts a sub-list of communications that satisfy the invariant. The rest represents the delayed communications

We stress the fact that all these functions are generic: their essential properties, called *proof obligations* or simply *constraints*, are formalized, but not their explicit definition.

3.2 Unfolding Function *GeNoC*

Function *GeNoC* is pictured on Fig. 3. It takes as arguments the list of requested communications and the characteristics of the network. It produces two lists as results: the messages received by the destination of successful communications and the aborted communications. In the remainder of this section, we detail the basic components of the model.

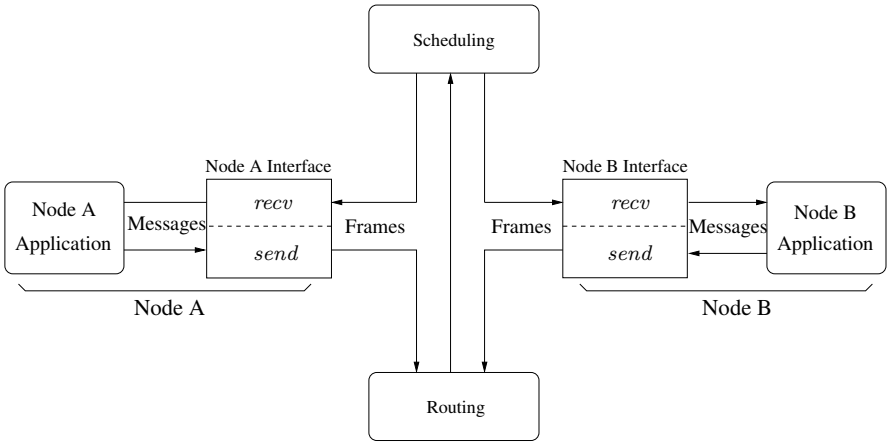


Fig. 3. *GeNoC*: A Generic Network

The main input of *GeNoC* is a list $\mathcal{T}$ of *transactions* of the form $t = (id\ A\ msg_t\ B)$. Transaction t represents the intention of application A to send a message msg_t to application B. A is the *origin* and B the *destination*. Both A and B are members of the set of nodes, *NodeSet*. Each transaction is uniquely identified by a natural *id*. Valid transactions are recognized by predicate $\mathcal{T}_{lstp}(\mathcal{T}, NodeSet)$.

The unfolding of function *GeNoC* is depicted in Figure 4. For every message in the initial list of transactions, function *ComputeMissives* applies function *send* to compute the corresponding frame. Each frame together with its *id*, *origin* and *destination* constitutes a *missive*. A missive is valid if the ids are naturals (with no duplicate); the origin and the destination are members of *NodeSet*. A valid list, $\mathcal{M}$ of missives is recognized by predicate $\mathcal{M}_{lstp}(\mathcal{M}, NodeSet)$. Then, function *Routing* computes a list of routes for every missive. If the routing algorithm is deterministic, this list has only one element. Once routes are computed, a *travel* denotes the list composed of a frame, its *id* and its list of routes. A list $\mathcal{V}$ of travels is valid if the ids are naturals (with no duplicate). Such a list is recognized by predicate $\mathcal{V}_{lstp}(\mathcal{V})$. Function *Scheduling* separates $\mathcal{V}$ into a list *Scheduled* of scheduled travels and a list *Delayed* of delayed travels. The results of the scheduled travels are computed by calling *recv*. Delayed travels are converted back to missives and constitute the argument of a recursive call to *GeNoC*.

To make sure that this function terminates, we associate a *finite* number of attempts to every node. At every recursive call of $GeNoC$, every node with a pending transaction consumes one attempt. The *association list* att stores the attempts and $att[i]$ denotes the number of remaining attempts for the node i. Function $SumOfAtt(att)$ computes the sum of the remaining attempts for all the nodes and is used as the decreasing measure of parameter att. Function $GeNoC$ halts if all attempts have been consumed.

The first output list $\mathcal{R}$ contains the results of the completed transactions. Every result r is of the form *(id B msg_r)* and represents the reception of a message msg_r by its final destination B. Transactions may not run to completion (*e.g.* due to network contention). The second output list of $GeNoC$ is named *Aborted* and contains the cancelled transactions.

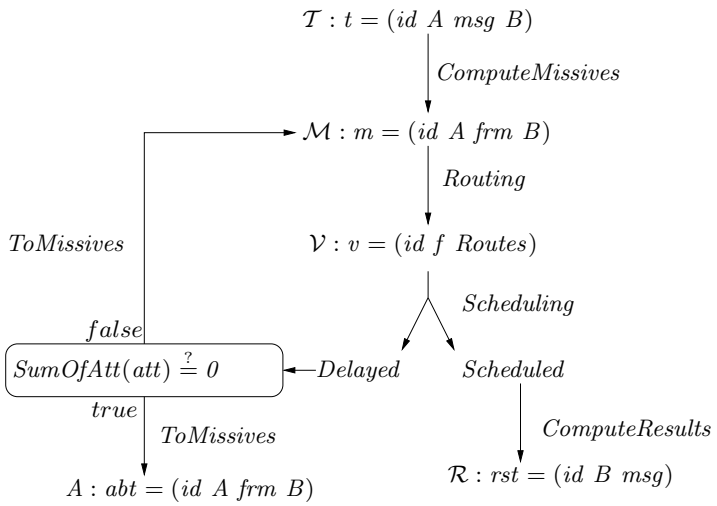


Fig. 4. Unfolding of function $GeNoC$

The correctness of $GeNoC$ is expressed by two properties. First, the messages that are received are identical to the messages that were sent. Second, each message is received by its expected destination. Formally, this is expressed by the formula below, which shows that each result rst is obtained from a unique transaction t that has the same identifier, the same message and the same destination as rst.

$$\forall rst \in_l \mathcal{R}, \exists! t \in_l \mathcal{T}, \begin{cases} Id_{\mathcal{R}}(rst) = Id_{\mathcal{T}}(t) \\ \wedge\ Msg_{\mathcal{R}}(rst) = Msg_{\mathcal{T}}(t) \\ \wedge\ Dest_{\mathcal{R}}(rst) = Dest_{\mathcal{T}}(t) \end{cases} \tag{4}$$

4 Details of the Functional Model

4.1 Nodes and Parameters

Nodes are defined on an arbitrary domain, *GenNodeSet*, with characteristic function *ValidNodep*:

$$\forall x,\ ValidNodep(x) \Leftrightarrow x \in GenNodeSet \tag{5}$$

The set of nodes of a particular network is noted *NodeSet*. In all this section, we shall use a subscripted curly $\mathcal{D}$ to represent a domain of elements. For instance, $\mathcal{D}_{msg}$ is the domain of messages, $\mathcal{D}_{frm}$ is the domain of frames, etc.

4.2 Interfaces

Function *send* builds a *frame* from a *message* and function *recv* builds a *message* from a *frame*. Their functionality is:

$$send : \mathcal{D}_{msg} \rightarrow \mathcal{D}_{frm} \tag{6}$$

$$recv : \mathcal{D}_{frm} \rightarrow \mathcal{D}_{msg} \tag{7}$$

The constraint on these functions is that their composition is the identity function. The following proof obligation has to be relieved:

Proof Obligation 1 Validity of The Interface Functions.

$$\forall msg \in \mathcal{D}_{msg}, recv \circ send(msg) = msg$$

4.3 Routing

Principles and correctness criteria Let d be the destination of a frame standing at node s. In the case of deterministic algorithms, the routing logic of a network selects a unique node as the next step in the route from s to d. This logic is represented by function $\mathcal{L}(s,d)$. The list of the visited nodes for every travel from s to d is obtained by the successive applications of function $\mathcal{L}$ until the destination is reached, i.e. while $\mathcal{L}(s,d) \neq d$. The route from s to d is:

$$s, \mathcal{L}(s,d), \mathcal{L}(\mathcal{L}(s,d),d), \mathcal{L}(\mathcal{L}(\mathcal{L}(s,d),d),d), \ldots, d$$

A route is computed by function ρ_{det} that recursively applies function $\mathcal{L}$ from the source node to the destination node. Function ρ_{det} is defined as follows:

$$\rho_{det}(s,d) \triangleq \begin{cases} d & \textbf{if } s = d \\ s.\rho_{det}(\mathcal{L}(s,d),d) & \textbf{otherwise} \end{cases} \tag{8}$$

In the adaptive case, the routing logic offers at each intermediate node several "next" nodes. Several routes are possible between a source s and a destination

d. In that case, the routing algorithm is represented by function ρ_{ndet}, which computes *all* possible routes between nodes s and d.

To cover the general case, the routing algorithm is represented by function ρ, which takes as arguments a source node s and a destination node d. This function returns the list of the possible routes between s and d. Its functionality is the following, where $\mathcal{C}$ denotes a list of lists of nodes:

$$\rho : GenNodeSet \times GenNodeSet \rightarrow \mathcal{C} \tag{9}$$

Routing Termination Since function ρ is recursive, it must be shown to terminate, both to ensure the liveness of the network, and to be accepted by a proof assistant.

Let S be a set and $\prec_S$ be a total ordering relation on S. We recall that $(S, \prec_s)$ is a well-founded structure if any subset of S has a minimal element for $\prec_S$. Typically, the proof of termination of a function is done by showing that some *measure* on its parameters is decreasing on a well-founded structure for every recursive call of that function.

Let us return to the deterministic case and function ρ_{det}. Let $(S, \prec_S)$ be a well-founded structure (most often S is the set of naturals), and mes be a measure on S.

$$mes : GenNodeSet \times GenNodeSet \rightarrow S$$

To prove that ρ_{det} terminates, one needs to prove that the "governing" condition for the recursive call, namely $s \neq d$, implies that mes is decreasing. The following proof obligation has to be satisfied:

Proof Obligation 2 Termination Condition for ρ_{det}.

$$\forall s, d \in GenNodeSet, \exists mes : GenNodeSet \times GenNodeSet \rightarrow S,$$
$$s \neq d \Rightarrow mes(\mathcal{L}(s,d), d) \prec_S mes(s,d)$$

Routing Correctness The correctness of a route is defined according to a missive. A route r is correct with respect to a missive m if r starts with the origin of m, ends with the destination of m and every node of r belongs to the set of nodes of the network. Every correct route has at least two nodes. The following predicate defines these conditions:

Definition 1. ValidRoutep.

$$ValidRoutep(r, m, NodeSet) \triangleq \begin{cases} r[0] = Org_{\mathcal{M}}(m) \\ \wedge\ Last(r) = Dest_{\mathcal{M}}(m) \\ \wedge\ r \subseteq_l NodeSet \wedge Len(r) \geq 2 \end{cases}$$

Whether routing is deterministic or adaptive, this predicate must be satisfied by all routes produced by function ρ. The following proof obligation has to be relieved:

Proof Obligation 3 Correctness of routes produced by ρ.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet)$$
$$\Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \rho(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)), ValidRoutep(r, m, NodeSet)$$

Definition and Validation of Function *Routing* Function *Routing* takes as arguments a missive list and the set *NodeSet* of nodes of the network. It returns a travel list in which a list of routes is associated to each missive. The functionality of *Routing* is the following:

$$Routing : \mathcal{D}_{\mathcal{M}} \times \mathcal{P}(GenNodeSet) \rightarrow \mathcal{D}_{\mathcal{V}} \tag{10}$$

Function *Routing* builds a travel list from the identifier, the frame, the origin and the destination of missives.

Definition 2. Function Routing
$Routing(\mathcal{M}, NodeSet) \triangleq$

$$\bigwedge_{m \in_l \mathcal{M}} List(Id_{\mathcal{M}}(m), Frm_{\mathcal{M}}(m), \rho(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)))$$

Concerning data types, one has to prove that function *Routing* produces a valid travel list if the initial missive list is valid.

Proof Obligation 4 Type of *Routing*.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow \mathcal{V}_{lstp}(Routing(\mathcal{M}, NodeSet))$$

The definition of function *Routing* preserves the properties proved about the previous function ρ. Function *Routing* terminates and the routes of every travel satisfy predicate *ValidRoutep*. In a missive list, identifiers are unique. For every travel v produced by function *Routing*, there is a unique missive m such that its identifier equals the identifier of v and the frame of v equals the frame of m.

Theorem 1. Missive/Travel Match.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow$$
$$\forall v \in_l Routing(\mathcal{M}, NodeSet), \exists! m \in_l \mathcal{M}, \begin{cases} Id_{\mathcal{V}}(v) = Id_{\mathcal{M}}(m) \\ \wedge\ Frm_{\mathcal{V}}(v) = Frm_{\mathcal{M}}(m) \end{cases}$$

Proof. By definition of *Routing*.

Travels delayed by the scheduling function - but produced by function *Routing* - are converted back to missives by function *ToMissives*. The latter builds missives in the following manner. It takes the identifier and the frame of a travel. The origin and the destination of a missive are the first and the last node of a route. Function *ToMissives* is the reverse of function *Routing*.

Theorem 2. Routing ToMissives.

$$\forall \mathcal{M}, \mathcal{M}_{lstp} \Rightarrow ToMissives \circ Routing(\mathcal{M}, NodeSet) = \mathcal{M}$$

Proof. Frames are not modified by function *Routing*. Since the latter satisfies predicate *ValidRoutep* for all routes of all travels that it produces, the first and the last node of any route are equal to the origin and the destination of the initial missive.

4.4 Scheduling

Function *Scheduling* takes as arguments the travel list produced by function *Routing* and the list *att* of the remaining number of attempts. It updates *att* and returns two travel lists: the list *Scheduled* and the list *Delayed*. The functionality of *Scheduling* is:

$$Scheduling : \mathcal{D}_{\mathcal{V}} \times AttLst \rightarrow \mathcal{D}_{\mathcal{V}} \times \mathcal{D}_{\mathcal{V}} \times AttLst \tag{11}$$

A scheduled travel only keeps one of the possible routes for the missive. For technical reasons, we avoid the introduction of a new data type and do not make a special case of scheduled travels: they contain a list of routes, even if this list has only one element.

The validation of *Scheduling* requires the satisfaction of several proof obligations.

In the following, the projection of a vector on one of its dimensions is denoted π_i^j, with the following functionality:

$$\pi_i^j : \mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_j \rightarrow \mathcal{D}_i \tag{12}$$

For instance, $\pi_1^2(x_1, x_2) = x_1$ and $\pi_2^2(x_1, x_2) = x_2$.

First, if the first parameter $\mathcal{V}$ of *Scheduling* is a valid travel list, the lists *Scheduled* and *Delayed* are also valid.

Proof Obligation 5 Type of *Scheduled* **and** *Delayed*.
Let Scheduled be $\pi_1^3 \circ Scheduling(\mathcal{V}, att)$ *and*
Delayed be $\pi_2^3 \circ Scheduling(\mathcal{V}, att)$, *then* :

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \mathcal{V}_{lstp}(Scheduled) \wedge \mathcal{V}_{lstp}(Delayed)$$

At each scheduling round, all travels of $\mathcal{V}$ are analyzed. If several travels are associated to a single node, this node consumes one attempt for the set of its travels. At each call to *Scheduling*, an attempt is consumed at each node. If all attempts have not been consumed, the sum of the remaining attempts after the application of function *Scheduling* is strictly less than the sum of the attempts before the application of *Scheduling*. This is expressed by the following proof obligation:

Proof Obligation 6 Function *Scheduling* **consumes at least one attempt.**
Let natt be $\pi_3^3 \circ Scheduling(\mathcal{V}, att)$, *then:*

$$\begin{aligned} & SumOfAtt(att) \neq 0 \\ \rightarrow\ & SumOfAtt(natt) < SumOfAtt(att) \end{aligned}$$

The list of the delayed travels must be a sublist of $\mathcal{V}$. Formally, one ensures that for every delayed travel dtr, there exists a unique initial travel v such that dtr and v have the same identifier, the same frame and the same routes. Hence the following proof obligation:

Proof Obligation 7 Correctness of the delayed travels.
Let $Delayed$ *be* $\pi_2^3 \circ Scheduling(\mathcal{V}, att)$*, then:*

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \forall dtr \in_l Delayed, \exists! v \in_l \mathcal{V}, \begin{cases} Id_{\mathcal{V}}(dtr) = Id_{\mathcal{V}}(v) \\ \wedge\ Frm_{\mathcal{V}}(dtr) = Frm_{\mathcal{V}}(v) \\ \wedge\ Routes_{\mathcal{V}}(dtr) = Routes_{\mathcal{V}}(v) \end{cases}$$

Since the scheduling function only keeps one route for every scheduled travel, the list $Scheduled$ is not exactly a sublist of the initial travel list $\mathcal{V}$. The identifiers and the frames are not modified. We check that the route, or more generally, the routes of a scheduled travel belong to the routes of the corresponding initial travel. Formally, we ensure that for every scheduled travel str, there exists a unique initial travel v such that str and v have the same identifier, the same frame and that the routes associated with str are among the routes associated with v.

Proof Obligation 8 Correctness of the scheduled travels.
Let $Scheduled$ *be* $\pi_1^3 \circ Scheduling(\mathcal{V}, att)$*, then:*

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \forall str \in_l Scheduled, \exists! v \in_l \mathcal{V}, \begin{cases} Id_{\mathcal{V}}(str) = Id_{\mathcal{V}}(v) \\ \wedge\ Frm_{\mathcal{V}}(str) = Frm_{\mathcal{V}}(v) \\ \wedge\ Routes_{\mathcal{V}}(str) \sqsubseteq Routes_{\mathcal{V}}(v) \end{cases}$$

Since routes of travels in $Scheduled$ are routes of travels of $\mathcal{V}$, function $Scheduling$ preserves the correctness of routes. If routes of $\mathcal{V}$ satisfy predicate $ValidRoutep$, so do the routes of $Scheduled$.

A travel cannot, at the same time, be scheduled and delayed.

Proof Obligation 9 Mutual exclusion between $Delayed$ **and** $Scheduled$**.**
Let $Scheduled$ *be* $\pi_1^3 \circ Scheduling(\mathcal{V}, att)$ *and*
$Delayed$ *be* $\pi_2^3 \circ Scheduling(\mathcal{V}, att)$*, then* :

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow Delayed_{\lfloor id} \sqcap Scheduled_{\lfloor id} = \epsilon$$

4.5 Definition and Validation of *GeNoC*

The definition of function $GeNoC$ and its correctness proof are summarized in Fig. 5. The recursive call in $GeNoC$ only involves functions $Routing$ and $Scheduling$. We define function $GeNoC_t$ to be the subfunction computing this recursion. It takes as arguments a list $\mathcal{M}$ of missives, the set $NodeSet$ of nodes of the network, the list att of the attempt numbers and a travel list $\mathcal{V}$ that is initially empty. It returns two lists: a travel list that contains the frames received

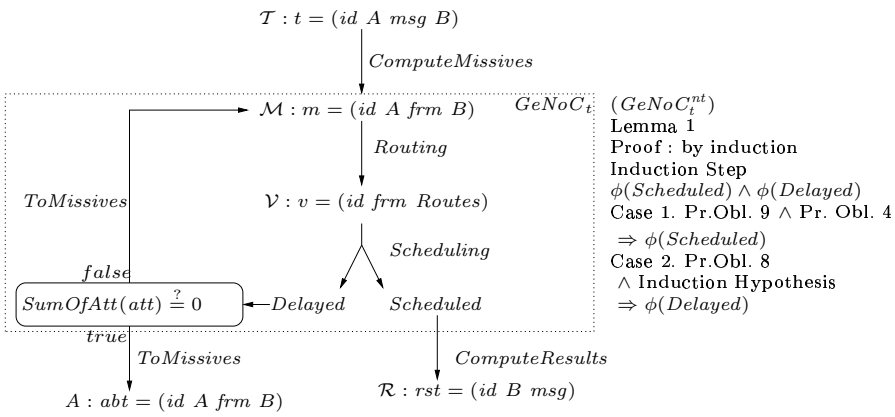


Fig. 5. Proof of $GeNoC$

by the destination nodes of the missives in $\mathcal{M}$ and a list that contains the aborted missives. Its functionality is the following:

$$GeNoC_t : \mathcal{D}_{\mathcal{M}} \times \mathcal{P}(GenNodeSet) \times AttLst \times D_{\mathcal{V}} \rightarrow \mathcal{D}_{\mathcal{V}} \times \mathcal{D}_{\mathcal{M}} \qquad (13)$$

If all attempts have been consumed, $GeNoC_t$ returns the travels accumulated in $\mathcal{V}$ and the list of the remaining missives, *i.e.* the aborted missives. Otherwise, the travels produced by function *Routing* are passed to function *Scheduling*. The scheduled travels are added to the list $\mathcal{V}$. The delayed travels are converted to missives and constitute an argument of the recursive call to $GeNoC_t$. The remaining arguments are the updates of the lists att and $\mathcal{V}$.

Definition 3. Definition of $GeNoC_t$.
$GeNoC_t(\mathcal{M}, NodeSet, att, \mathcal{V}) \triangleq$
if $SumOfAtt(att) = 0$ **then**
$List(\mathcal{V}, \mathcal{M})$
else
Let$(ScheduledRtg\ DelayedRtg\ att_1)$ **be**
$Scheduling(Routing(\mathcal{M}, NodeSet), att)$ **in**
$GeNoC_t(ToMissives(DelayedRtg), NodeSet, att_1, ScheduledRtg \sqcup \mathcal{V})$
endif

The correctness of function $GeNoC_t$ is obtained if for every element ctr of the completed travels G, the frame and the last node of the route [2] of ctr are equal to the frame and the destination of the missive m in $\mathcal{M}$ that has the same identifier as ctr. This is expressed by the lemma below:

[2] Note that to keep our notations consistent, a travel is always made of a list of routes, even if this list has only one element.

Lemma 1. Correctness of $GeNoC_t$.

$$\forall ctr \in_l G, \exists! m \in_l \mathcal{M}, \begin{cases} Id_{\mathcal{V}}(ctr) = Id_{\mathcal{M}}(m) \\ \wedge\ Frm_{\mathcal{V}}(ctr) = Frm_{\mathcal{M}}(m) \\ \wedge\ \forall r \in_l Routes_{\mathcal{V}}(ctr), Last(r) = Dest_{\mathcal{M}}(m) \end{cases}$$

Where:

$$G = \pi_1^2 \circ GeNoC_t(\mathcal{M}, NodeSet, att, \epsilon)$$

Proof. This theorem is proven by induction on the structure of function $GeNoC_t$. It follows from proof obligation 9 that the scheduled and the delayed travels can be proven separately. Scheduled travels have a correspondance with the travel list input in *Scheduling* (proof obligation 8). Function *Routing* produces correct routes (proof obligation 3), which are still correct after *Scheduling*. So, frames and destinations after *Scheduling* match the missives input to function *Routing*. The delayed travels are proven using the induction hypothesis and proof obligation 7.

Function $GeNoC$ takes as arguments a list $\mathcal{T}$ of transactions, the set $NodeSet$ of nodes of the network, the list att of attempt numbers. It returns the list $\mathcal{R}$ containing the results and the list A containing the aborted missives. It has the following functionality:

$$GeNoC : \mathcal{D}_{\mathcal{T}} \times \mathcal{P}(GenNodeSet) \times AttLst \rightarrow \mathcal{D}_{\mathcal{R}} \times \mathcal{D}_{\mathcal{M}} \tag{14}$$

Function *ComputeMissives* applies function *send* to the message of each transaction of the list $\mathcal{T}$. This function produces a list of missives from the initial transactions. Its functionality is the following:

$$ComputeMissives : \mathcal{D}_{\mathcal{T}} \rightarrow \mathcal{D}_{\mathcal{M}} \tag{15}$$

It is defined as follows:

Definition 4. ComputeMissives.
$ComputeMissives(\mathcal{T}) \triangleq$

$$\bigwedge_{t \in_l \mathcal{T}} List(Id_{\mathcal{T}}(t), Org_{\mathcal{T}}(t), send(Msg_{\mathcal{T}}(t)), Dest_{\mathcal{T}}(t))$$

Function *ComputeResults* applies function *recv* to each frame of a travel list to produce a list of results. Its functionality is the following:

$$ComputeResults : \mathcal{D}_{\mathcal{V}} \rightarrow \mathcal{D}_{\mathcal{R}} \tag{16}$$

It is defined as follows:

Definition 5. ComputeResults.
$ComputeResults(\mathcal{V}) \triangleq$

$$\bigwedge_{tr \in_l \mathcal{V}} List(Id_{\mathcal{V}}(tr), Last(Routes_{\mathcal{V}}(tr)), recv(Frm_{\mathcal{V}}(tr)))$$

Function $GeNoC$ is defined using these functions and $GeNoC_t$. Function $ComputeMissives$ gives the first argument of $GeNoC_t$ from the transaction list $\mathcal{T}$. The last argument of $GeNoC_t$ is the empty list. The aborted missives are produced by function $GeNoC_t$. The definition of $GeNoC$ is the following:

Definition 6. Definition of $GeNoC$**.**
$GeNoC(\mathcal{T}, NodeSet, att) \triangleq$
Let $(Responses\ Aborted)$ **be**
$GeNoC_t(ComputeMissives(\mathcal{T}), NodeSet, att, \epsilon)$ **in**
$List(ComputeResults(Responses), Aborted)$

The correctness of $GeNoC$ is defined by expression 4 defined in section 3.

Theorem 3. Correctness of $GeNoC$**.**
Let $\mathcal{R}$ *be* $\pi_1^2 \circ GeNoC(\mathcal{T}, NodeSet, att)$ *in*

$$\forall rst \in_l \mathcal{R}, \exists! t \in_l \mathcal{T}, \begin{cases} Id_{\mathcal{R}}(rst) = Id_{\mathcal{T}}(t) \\ \wedge\ Msg_{\mathcal{R}}(rst) = Msg_{\mathcal{T}}(t) \\ \wedge\ Dest_{\mathcal{R}}(rst) = Dest_{\mathcal{T}}(t) \end{cases}$$

Proof. The last term of the conjunct is directly obtained from Lemma 1. From this lemma, it also follows that the frames produced by function $ComputeMissives$ are identical to the frames converted in messages by function $ComputeResults$. From proof obligation 1 on the interfaces, it comes that messages of results are equal to messages in the initial transaction list.

5 Methodology and Case Studies

We have embedded our theory in the logic of the ACL2 theorem proving system [15]. Despite the fact that ACL2 is first order, and does not support the explicit use of quantifiers, the choice of this system offered a number of advantages:

- The input language being a subset of Common Lisp, the functions are executable. It is realistic to execute a model on test benches, and visualize the behavior of a particular network specification, as a first debugging step before proceeding with human time consuming proofs. This feature is important also for quick software prototyping, as a basis of discussion with network designers.
- A large number of existing previous works are publicly available, and developing a new theory benefits from many layers of expert developments that extend the system first principles. Libraries of functions definitions and proven theorems can be compiled and stored for later use, restoring an environment is a single statement.
- Very powerful definition mechanisms, such as the *encapsulation principle*, allow to extend the logic and reason on undefined functions that satisfy one or more theorems, provided one witness can be exhibited. We made an

extensive use of this principle to prove the correctness of *GeNoC* assuming the satisfaction of the constraints on the functions that formalize the network constituents.
– The combined use of typing predicates, list filtering, implication and recursive function definitions over list arguments provides a means to express universally quantified properties over domains, and the statement "there exists a unique element such that".

Applying a systematic, and reusable, mode of expression (see [25] for details), the complete *GeNoC* formalization could be performed in the ACL2 logic, with the above listed benefits, and we thus benefited from the high degree of automated mechanized reasoning in ACL2.

The proof of the main theorem about *GeNoC* and its modules involve 71 functions, 119 theorems in 1864 lines of code. Only one fourth of these is dedicated to the encapsulation of the different modules. Most of the definitions and theorems concern data types and the proof of the overall correctness. This makes *GeNoC* "relatively simple" to use, because users will only be concerned with the modules, as we shall now discuss.

5.1 Overview of the Applications

In Fig 6, we summarize concrete instances of *GeNoC*. Any combination of these different concrete instances is defined and validated by generic function *GeNoC*, that means without any additional effort.

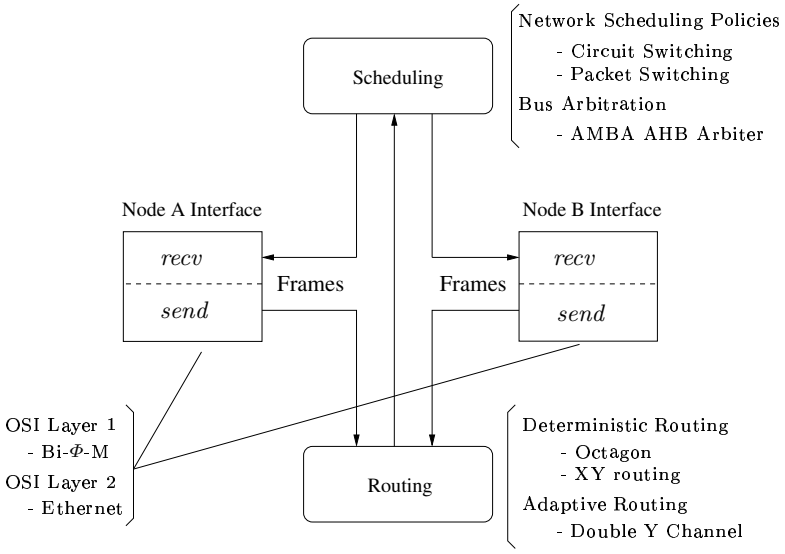


Fig. 6. Concrete Instances of *GeNoC*

We have shown that the circuit [23] and the packet [24] switching techniques are concrete instances of *Scheduling*. Based on previous work [22], we proved that bus arbitration in the AMBA AHB is also a valid instance of the generic scheduling policy. From Moore's work on asynchrony [17], we proved that his model of the biphase protocol constitutes a valid instance of the interfaces. We have modeled an Ethernet controler[3] and we are investigating its compliance with *GeNoC*.

In the next subsections, we illustrate our approach on the Octagon network. We first detail the methodology associated with the routing algorithm. Then, we apply it to the routing algorithm of the Octagon. A model and a proof of this network have already been presented [23], but with a different methodology. We have also shown that XY routing in a 2D mesh is also a valid instance of our generic model [24]. Finally, we are currently working on the proof that an adaptive routing algorithm - the double Y channel algorithm in a 2D mesh - is a valid instance of function *Routing*. More details about all these studies can be found in Schmaltz's thesis [21].

5.2 Concrete instances of function *Routing*

The topology of a network determines the node numbering and the unitary moves allowed between two adjacent nodes. The routing function is defined by the successive applications of these moves. Before defining a particular routing function, one has to define the set of nodes.

Node Definition. Before all, one has to define the node definition domain, that is a particular instance of predicate *ValidNodep*, noted $ValidNodep_\sharp$. The generic definition domain *GenNodeSet* becomes a particular domain $GenNodeSet_\sharp$, the naturals for instance. One has to give a concrete definition of Equation 5, that is:

$$\forall x,\ ValidNodep_\sharp(x) \Leftrightarrow x \in GenNodeSet_\sharp \tag{17}$$

Routing Definition First, we identify the moves allowed between two adjacent nodes. As we consider regular network (or a regularization of an irregular network), these moves are all identical at each point of the network. Identifying these unitary moves defines a concrete instance, $\mathcal{L}_\sharp$, of the routing logic $\mathcal{L}$. The routing function results of the successive application of these unitary moves, that is:

$$\rho_\sharp(s,d) \triangleq \begin{cases} d & \textbf{if } s = d \\ s.\rho_\sharp(\mathcal{L}_\sharp(s,d),d) & \textbf{otherwise} \end{cases} \tag{18}$$

The distance between the current position of a message and its destination is deduced from the topology. The distance between some node s and some node d is noted $dist(s,d)$. Most often, this distance is the measure used to prove that the routing function terminates. It suffices to prove that each unitary move reduces

[3] This work has been done during a visit of the first author at the University of Texas at Austin, in cooperation with Warren Hunt.

this distance. The distance is a function that returns a natural for any node pair. This function has the following functionality:

$$dist : GenNodeSet_\sharp \times GenNodeSet_\sharp \rightarrow \mathbb{N} \tag{19}$$

To prove the termination of the routing function, $\rho_\sharp$, one has to prove that this function satisfies a concrete instance of proof obligation 2:

$$\forall s, d \in GenNodeSet_\sharp, s \neq d \Rightarrow dist(\mathcal{L}_\sharp(s,d),d) < dist(s,d) \tag{20}$$

The validity of a route is tested by predicate *ValidRoutep*. The definition of *ValidRoutep* is valid for all networks, it needs not be redefined (see Definition 1).

Finally, to validate the concrete routing function, it suffices to prove that is satisfies predicate *ValidRoutep* for the set $NodeSet_\sharp$ of concrete nodes of the network:

$$\begin{array}{l}\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_\sharp) \\ \Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \rho_\sharp(Org_\mathcal{M}(m), Dest_\mathcal{M}(m)), ValidRoutep(r, m, NodeSet_\sharp)\end{array} \tag{21}$$

A function that matches the generic definition $Routing_\sharp$ computes a list of routes for each missive of a list $\mathcal{M}$:

Definition 7. Concrete Instance of Function *Routing*.
$Routing_\sharp(\mathcal{M}, NodeSet_\sharp) \triangleq$

$$\bigwedge_{m \in \mathcal{M}} List(Id_\mathcal{M}(m), Frm_\mathcal{M}(m), \rho_\sharp(Org_\mathcal{M}(m), Dest_\mathcal{M}(m)))$$

To prove the compliance of this function with *GeNoC*, we still need to prove that $Routing_\sharp$ produces a valid travel list if the initial list $\mathcal{M}$ is a valid list of missives:

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_\sharp) \Rightarrow \mathcal{V}_{lstp}(Routing_\sharp(\mathcal{M}, NodeSet_\sharp)) \tag{22}$$

We apply this methodology to the Octagon network presented in Section 2.1.

5.3 Octagon Case Study

Octagon Node Definition Our Octagon model considers an arbitrary, but finite, number of nodes, noted *NumNode*. This number is a natural, multiple of 4. So, we can define that number using a natural N, $NumNode = 4N$. Predicate $ValidNodep_{Oct}$ takes as arguments a node x and number N:

$$\forall N \in \mathbb{N}, \forall x, ValidNodep_{Oct}(x, N) \Leftrightarrow x \in \mathbb{N} \wedge x < 4N \tag{23}$$

Octagon Routing Function Let s be the current node and d the destination node. The three unitary moves in the Octagon are defined as:

$$Clockwise(s, NumNode) \triangleq (s+1) \bmod NumNode$$

$$CounterClockwise(s, NumNode) \triangleq (s-1) \bmod NumNode$$

$$Across(s, NumNode) \triangleq (s + \frac{NumNode}{2}) \bmod NumNode$$

These moves are grouped into function $\mathcal{L}_{Oct}$. The relative address is $RelAd = (d-s) \bmod 4N$. If the current node is the destination, the message is consumed. If the relative address is positive and less than N, the message moves clockwise. If this address is between $3N$ and $4N$, it moves counterclockwise. Otherwise, it moves across. The definition of $\mathcal{L}_{Oct}$ is as follows:

Definition 8. Unitary moves in the Octagon.

$$\mathcal{L}_{Oct}(s,d,N) \triangleq \begin{cases} s & \textbf{if } RelAd = 0 \\ Clockwise(s, 4N) & \textbf{if } 0 < RelAd \leq N \\ CounterClockwise(s, 4N) & \textbf{if } 3N \leq RelAd < 4N \\ Across(s, 4N) & \textbf{otherwise} \end{cases}$$

Routing function ρ_{Oct} is defined as the recursive application of the unitary moves:

Definition 9. Routing Function of the Octagon, ρ_{Oct}.

$$\rho_{Oct}(s,d,N) \triangleq \begin{cases} d & \textbf{if } s = d \\ s.\rho_{Oct}(\mathcal{L}_{Oct}(s,d,N), d, N) & \textbf{otherwise} \end{cases}$$

As there are two ways of traversing the Octagon, there exist two distances between two nodes. The measure used to prove that function ρ_{Oct} terminates is the minimum between these two distances:

$$mes_{Oct}(s, d, NumNode) = Min[(d-s) \bmod NumNode, (s-d) \bmod NumNode]$$

To prove that the octagon routing function terminate, it suffices to prove that the unitary moves reduce this distance:

Theorem 4. Octagon Routing Function Terminates.
$\forall s, d \in GenNodeSet_{Oct}, s \neq d \Rightarrow$

$$mes_{Oct}(\mathcal{L}_{Oct}(s,d), d, NumNode) < mes_{Oct}(s, d, NumNode)$$

Proof. The proof is decomposed according to the different moves. Each one of them reduces the distance. The proof is a huge case split because of functions Min and mod. In ACL2, the proof is decomposed in more that 1200 cases. It only requires 10 additional lemmas about function modulo in addition to the latest arithmetic library [18]. Two lemmas are also required to drive ACL2 to the right case split. The proof is automatically performed in less that 100 seconds on a Pentium IV 1,6 GHz, 256 MB of memory and running under Linux.

To show that function ρ_{Oct} constitutes a valid instance of the generic routing function, we need to prove that it produces routes which satisfy predicate *ValidRoutep*:

Theorem 5. Validity of Octagon Routes.

$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{Oct})$
$\Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \rho_{Oct}(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)), ValidRoutep(r, m, NodeSet_{Oct})$

Proof. By induction on the route length.

Finally, function $Routing_{Oct}$ follows the generic signature:

Definition 10. Octagon Routing, function $Routing_{Oct}$
$Routing_{Oct}(\mathcal{M}, NodeSet_{Oct}) \triangleq$

$$\bigwedge_{m \in \mathcal{M}} List(Id_{\mathcal{M}}(m), Frm_{\mathcal{M}}(m), List(\rho_{Oct}(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m))))$$

We still need to prove that this function produces a valid travel list. The proof of the following theorem is trivial:

Theorem 6. Type of Octagon Routes.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{Oct}) \Rightarrow \mathcal{V}_{lstp}(Routing_{Oct}(\mathcal{M}, NodeSet_{Oct}))$$

Table 5.3 shows details about the ACL2 modeling and proof. ACL2 is run on a Pentium IV at 1.6 GHz with 256 MB under Linux. The Octagon specification and proof are relatively small, an important point for the initial high level design step. In the proof, a huge amount of time is devoted to arithmetic reasoning.

	Nbr. of functions	Nbr. of theorems	Proof Time (seconds)	Size
OctagonNodeSet	5	4	< 1	70 lines
Lemmas on mod	0	10	< 3	150 lines
Routing	19	41	~ 720	955 lines
Total	21	64	< 740	1325 lines

Table 2. Functions, theorems and proof time for the definition and validation of the Octagon

6 Conclusion and Future Work

We have presented a generic model for communication architectures. It is formalized by function *GeNoC*, which is defined by three key components: interfaces, a routing algorithm and a scheduling policy. The generic model does not assume

any particular definition of these components. It only relies on a set of proof obligations (or constraints) associated with each component. The correctness of *GeNoC* includes the proof that messages are either lost or reach their expected destination without modification of their content. This proof is deduced from the proof obligations only. Hence, the specification and the validation of a particular communication architecture amounts to give an *explicit* definition to each component and to prove that these definitions satisfy the corresponding constraints. Moreover, each component is self-contained and can be specified and validated in isolation.

To validate our approach, we have applied it to a variety of architectures that constitute as many concrete instances of our theory: some come from industrial systems, like the AMBA bus or the Octagon network, others are more academic examples, like XY or double Y channel routing in a 2D mesh, packet and circuit switching techniques or the biphase mark protocol Bi-ϕ-M.

The current *GeNoC* definition is very abstract and very simplified. Successive, proven correct refined models are needed before reaching the level of details of an implementation specification. Our work is currently being extended in two different directions. At TIMA, our research involves the application of *GeNoC* to wormhole routing, and the elaboration of a refinement method to derive the correctness of a particular hardware implementation. In Germany, the Verisoft [1] project aims at developing methods and tools for the pervasive verification of computer systems. Theories have already been developed about processors [4], operating systems [9], compilers [16], memories [7], and I/O devices [13]. We aim at verifying a complete distributed system where each node will contain each one of the above components and where nodes are connected through a time triggered bus in a FlexRay flavor. A pencil and paper proof of such a system already exists [3]. From this proof, we have sketched additional constraints on the interfaces to ensure proper communication in a real time context [20]. Ultimately, *GeNoC* will be used as the integration of the different theories in a single framework.

Acknowledgment

The authors would like to thank J Strother Moore, Matt Kaufmann and Warren Hunt for valuable remarks and helpful advices regarding ACL2. We are also thankful to Katell Morin-Allory for suggesting improvements to a previous version of this paper.

References

1. http://www.verisoft.de.
2. H. Amjad. Model Checking the AMBA Protocol in HOL. Technical report, University of Cambridge, Computer Laboratory, September 2004.
3. S. Beyer, P. Böhm, M. Gerke, M. Hillebrand, T. In der Rieden, S. Knapp, D. Leinenbach, and W.J. Paul. Towards the formal verification of lower system layers in automotive systems. In *23nd IEEE International Conference on Computer Design:*

VLSI in Computers and Processors (ICCD 2005), 2-5 October 2005, San Jose, CA, USA, Proceedings, pages 317–324. IEEE, 2005.
4. S. Beyer, C. Jacobi, D. Kroning, D. Leinenbach, and W.J. Paul. Instantiating Uninterpreted Functional Units and Memory System: Functional Verification of the VAMP. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *LNCS*, pages 51–65, L'Aquila, Italy, October 2003. Springer-Verlag.
5. R. S. Boyer and J Strother Moore. *A Computation Logic Handbook*. Academic Press, 1988.
6. W. Büttner. Is Formal Verification Bound to Remain a Junior Partner of Simulation? In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, 2005. Invited Speaker.
7. I. Dalinger, M. Hillebrand, and W.J. Paul. On the Verification of Memory Management Mechanisms. In D. Borrione and W.J. Paul, editors, *CHARME 2005*, LNCS. Springer, 2005.
8. W.J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan-Kaufmann Publisher, 2004.
9. M. Gargano, M. Hillebrand, D. Leinenbach, and W.J. Paul. On the Correctness of Operating System Kernels. In J. Hurd and T. Melham, editors, *TPHOLs 2005*, LNCS. Springer, 2005.
10. B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Rădulescu. Deadlock Prevention in the Æthereal protocol. In D. Borrione and W.J. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 345–348, 2005.
11. K. Goossens, J. Dielissen, and A. Rădulescu. Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers*, 22(5):414–421, September-October 2005.
12. D. Herzberg and M. Broy. Modeling Layered Distributed Communication Systems. *Formal Aspects of Computing*, 17(1):1–18, 2005.
13. M. Hillebrand, T. In der Rieden, and W.J. Paul. Dealing with I/O Devices in the Context of Pervasive System Verification. In *23nd IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 2005), 2-5 October 2005, San Jose, CA, USA, Proceedings*, pages 309–316. IEEE, 2005.
14. K. Karim, A. Nguyen, and S. Dey. An Interconnect Architecture for Networking Systems On Chip. *IEEE Micro*, pages 36–45, September-October 2002.
15. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *ACL2 Computer Aided Reasoning: An Approach*. Klulwer Academic Press, 2000.
16. D. Leinenbach, W.J. Paul, and E. Petrova. Towards the Formal Verification of a C0 Compiler: Code Generation and Implementation Correctness. In *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5-9 September 2005, Koblenz, Germany*, 2005.
17. J Strother Moore. A Formal Model of Asynchronous Communications and Its Use in Mechanically Verifying a Biphase Mark Protocol. *Formal Aspects of Computing*, 6(1):60–91, 1993.
18. W. A. Hunt R. Krug and J Strother Moore. Linear and Nonlinear Arithemetic in ACL2. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *LNCS*, pages 51–65, L'Aquila, Italy, October 2003. Springer-Verlag.
19. A. Roychoudhury, T. Mitra, and S.R. Karri. Using Formal Techniques to Debug the AMBA System-on-Chip Bus Protocol. In *Design Automation and Test Europe (DATE'03)*, pages 828–833, 2003.

20. J. Schmaltz. A Formal Model of Lower System Layer. In *Formal Methods in Computer-Aided Design (FMCAD'06)*, San Jose, CA, USA, November 12-16 2006. IEEE/ACM. (To appear).
21. J. Schmaltz. *Une formalisation fonctionnelle des communications sur la puce.* PhD thesis, Joseph Fourier University, Grenoble, France, January 2006. In French. A partial translation is available upon request to the first author.
22. J. Schmaltz and D. Borrione. Verification of a Parameterized Bus Architecture Using ACL2. In *Proceedings of the Fourth International Workshop on the ACL2 Theorem Prover and its Applications*, April 2003.
23. J. Schmaltz and D. Borrione. A Functional Approach to the Formal Specification of Networks on Chip. In A.J. Hu and A.K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 52–66, Austin, Tx, USA, November 2004. Springer-Verlag.
24. J. Schmaltz and D. Borrione. A Generic Network on Chip Model. In T. Melham and J. Hurd, editors, *Theorem Proving in Higher Order Logics (TPHOLs'05)*, volume 3603 of *LNCS*, pages 310–325, Oxford, UK, August 2005. Springer-Verlag.
25. J. Schmaltz and D. Borrione. Towards a Formal Theory of On Chip Communications in the ACL2 Logic. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, San Jose, California, USA, August 2006. ACM.
26. G. Spirakis. Beyond Verification: Formal Methods in Design. In A. Hu and A.K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, Austin, Texas, USA, November 2004. Springer-Verlag. Invited Speaker.

Isolating Intrusions by Automatic Experiments

Stephan Neuhaus
Lehrstuhl für Softwaretechnik
Universität des Saarlandes
Stephan.Neuhaus@acm.org

April 13, 2006

1 Introduction

The analysis of security incidents remains one of the most taxing things a computer scientist can do. Why is there no automated support for this task? We think this is so because existing tools use an inadequate methodology.

Intrusion analysis aims at reconstructing the break-in based on the current state of the system. To this end, we analyze traces and then deduce what must have happened inside the system so that these traces appear the way they do. For example, an analysis of the Linux Slapper worm could look like this: "Attackers with the IP address 10.120.130.140 sent a specially crafted HTTP request to our web server, which contained a malformed client key. This caused a buffer overflow and called a shell. This shell then saved a uuencode-encoded copy of the work source code, decoded and compiled it, and started the resulting program under the name *.bugtraq*. As soon as the program ran, it tried to contact other hosts i the network." (Example taken from [5].)

An investigator analyzing this intrusion will probably first see the rogue *.bugtraq* process and will then try to isolate those processes that were responsible for the attack. This holds for processes that are still running (such as the web server) and processes that have already terminated (such as the *uudecode* process).

The usual method is to begin with the violation of the security policy (the *.bugtraq* process) and then work backwards using tools like The Coroner's Toolkit [2] to the root cause (the malformed HTTPS request). This deductive approach has a number of serious drawbacks:

Completeness. The traces may not be sufficient in order to deduce the cause-effect chain reliably.

Minimality. Important traces are often buried in a large number of irrelevant traces and need to be laboriously extraced.

Correctness. Our proofs could base on wrong assumptions which may invalidate our deductions.

We have developed a tool called Malfor (short for MALware FORensics) which avoids these drawbacks by using *experimental* methods. Instead of interpreting traces and deducing a cause-effect chain backwards, Malfor works experimentally: in a first phase, Malfor *captures* events (processes in pur case) as the system is running. As soon as a break-in is detected, Malfor uses these events to *partially replay* the system. By cleverly choosing which events to repeat, we isolate those events that are reevant for the break-in: if we repeat the system without process X and if the break-in still occurs, process X cannot have been relevant for the attack.

2 Capture and Replay

Malfor's subsystem for capture and replay works by System Call Interposition. In this method, system calls like *fork*, *execve*, *read*, *getpid* and so on are diverted to Malfor's own routines. These execute the original routines and upload the system calls' parameters and results to a database. In security research, this method has been used in Systrace [6] in order to create on-the-fly security policies for system calls.

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/696

Malfor must take care of many details when replaying system calls, because otherwise replay will not work. For example, processes may have a different process ID during replay than it had during capturing. Still, the process must see its original PID so that library calls that use the PID (such as *gethostbyname*) still work as expected when replayed.

Our method works by replaying captured processes in ever different configurations. For this it is necessary that Malfor be able to suppress a process's execution. But on one hand, you can't force a parent process *not* to call *fork*. On the other hand, process creation must not simply fail because this would be too strong a difference with respect to the original run. Our solution is to create the child process, but to terminate it again at the next syscall.

These measures are typical when one wants to repeat only parts of a system.

3 Minimization

In order to find the responsible processes among all captured processes, we use Delta Debugging [3]. Delta Debugging is a technique that uses repeated experiments to minimize *any* set of failure-inducing circumstances.

Delta debugging works like binary search: first, we try with one half of all circumstances removed. If that reproduces the failure, we continue with this reduced set of circumstances. If not, however, we try by removing the other half. If that doesn't work either, we try the complements ofour subsets. If that doesn't work either, we split the original set into more than two parts and try again.

Zeller and others have shown that the final result contains only circumstances that are relevant for the failure. If there are initially n circumstances, delta debugging will need at most $O(n^2)$ tests to minimize them.

4 First Experiences

In order to test our prototype, we have witten a network server that contains a security hole: once it receives a specially prepared request, it creates a file */tmp/pwned* with administrator privileges. In a simulated attack, we have hidden one malicious request among twenty-nine others.

This run caused about 1,500 system calls, which were executed and captured by the original system in about 6 seconds. This is a performance overhead of about 8% with respect to the throughput without capturing. Capturing takes place in a virtual machine in order to simplify replay. Takting that into account as well, the overhead rises to 13% with respect to a dedicated machine. These penalties compare favourably with other research [1] and make Malfor suitable for production environments.

Malfor used about three minutes and 14 tests to isolate all relevant processes (three of 32) [5]. Replay was slower than capturing by a factor of about two. These numbers emphasize Malfor's suitability for production use.

5 Further Work

We first want to extend Malfor to a realistic example. We have already prepared an attack on Apache which adds another root account to the password file without opening the password file for reading. This attack is constructed especially to fool tools like BackTracker which analyze attacks by constructing relationships between system calls [1, 4]. This attack never opens the password file; yet it is modified afterwards.

The next task is to extend Malfor to distributed systems. Malfor is already designed to be used in such environments, but replaying needs to observe certain constraints so that the consistency of the entire system is preserved.

6 Conclusion

We have introduced Malfor, a system that uses experimental methods to analyse intrusions automatically. It can be used on production systems and is especially suitable for the analysis of targeted attacks.

Literatur

[1] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementati-*

on, pages 211–224, New York, NY, USA, December 2002. ACM Press.

[2] Dan Farmer. Frequently asked questions about the coroner's toolkit. `http://www.fish.com/tct/FAQ.html`, January 2005.

[3] Ralf Hildebrandt and Andreas Zeller. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 26(2):183–200, February 2002.

[4] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 223–236, New York, NY, USA, 2003. ACM Press.

[5] Stephan Neuhaus and Andreas Zeller. Isolating intrusions by automatic experiments. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, pages 71–80, Reston, VA, USA, February 2006. Internet Society, Internet Society.

[6] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th Usenix Security Symposium*, pages 257–272, Berkeley, CA, USA, August 2003. Usenix Association, Usenix Association.

Relating two standard notions of secrecy *

Eugen Zălinescu, Véronique Cortier, and Michaël Rusinowitch

Loria UMR 7503 & INRIA Lorraine projet Cassis & CNRS, France

Abstract. Two styles of definitions are usually considered to express that a security protocol preserves the confidentiality of a data `s`. Reachability-based secrecy means that `s` should never be disclosed while equivalence-based secrecy states that two executions of a protocol with distinct instances for `s` should be indistinguishable to an attacker. Although the second formulation ensures a higher level of security and is closer to cryptographic notions of secrecy, decidability results and automatic tools have mainly focused on the first definition so far.
This paper initiates a systematic investigation of situations where syntactic secrecy entails strong secrecy. We show that in the passive case, reachability-based secrecy actually implies equivalence-based secrecy for signatures, symmetric and asymmetric encryption provided that the primitives are probabilistic. For active adversaries in the case of symmetric encryption, we provide sufficient (and rather tight) conditions on the protocol for this implication to hold.

1 Introduction

Cryptographic protocols are small programs designed to ensure secure communications. Since they are widely distributed in critical systems, their security is primordial. In particular, verification using formal methods attracted a lot of attention during this last decade. A first difficulty is to formally express the security properties that are expected. Even a basic property such as confidentiality admits two different acceptable definitions namely reachability-based (*syntactic*) secrecy and equivalence-based (*strong*) secrecy. Reachability-based secrecy is quite appealing: it says that the secret is never accessible to the adversary. For example, consider the following protocol where the agent A simply sends a secret s to an agent B, encrypted with B's public key.

$$A \to B : \{\mathtt{s}\}_{\mathrm{pub}(B)}$$

An intruder cannot deduce `s`, thus `s` is syntactically secret. Although this notion of secrecy may be sufficient in many scenarios, in others, stronger security requirements are desirable. For instance consider a setting where `s` is a vote and B behaves differently depending on its value. If the actions of B are observable, `s` remains syntactically secret but an attacker can learn the values of the

* This work appeared as the INRIA research report RR-5908 (available at http://www.inria.fr/rrrt/rr-5908.html).

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/691

vote by watching B's actions. The design of equivalence-based secrecy is targeted at such scenarios and intuitively says that an adversary cannot observe the difference when the value of the secret changes. This definition is essential to express properties like confidentiality of a vote, of a password, or the anonymity of participants to a protocol.

Although the second formulation ensures a higher level of security and is closer to cryptographic notions of secrecy, so far decidability results and automatic tools have mainly focused on the first definition. The syntactic secrecy preservation problem is undecidable in general [12], it is co-NP-complete for a bounded number of sessions [16], and several decidable classes have been identified in the case of an unbounded number of sessions [12, 9, 7, 17, 15]. These results often come with automated tools, we mention for example ProVerif [5], CAPSL [11], and Avispa [4]. To the best of our knowledge, the only tool capable of verifying this property is the resolution-based algorithm of ProVerif [6] that has been extended to strong secrecy and only one decidability result is available [13]. In this article, Hüttel proves decidable for a fragment of the spi-calculus without recursion for framed bisimilarity, a related equivalence relation introduced by Abadi and Gordon [2].

In light of the above discussion, it may seem that the two notions of secrecy are separated by a sizable gap from both a conceptual but also from a practical point of view. These two notions have counterparts in the cryptographic setting (where messages are bitstrings and the adversary is any polynomial probabilistic Turing machine). Intuitively, the syntactic secrecy notion can be translated into a similar reachability-based secrecy notion and equivalence-based notion is close to indistinguishability. A quite surprising result [10] states that cryptographic syntactic secrecy actually implies indistinguishability in the cryptographic setting. This result relies in particular on the fact that the encryption schemes are probabilistic thus two encryptions of the same plaintext lead to different ciphertexts.

Motivated by the result of [10] and the large number of available systems for syntactic secrecy verification, we initiate in this paper a systematic investigation of situations where syntactic secrecy entails strong secrecy. Surprisingly, this happens in many interesting cases.

We offer results in both passive and active cases in the setting of the *applied pi calculus* [1]. We first treat in Section 2 the case of passive adversaries. We prove that syntactic secrecy is equivalent to strong secrecy. This holds for signatures, symmetric and asymmetric encryption. It can be easily seen that the two notions of secrecy are not equivalent in the case of deterministic encryption. Indeed, the secret $\mathtt{s}$ cannot be deduced from the encrypted message $\{\mathtt{s}\}_{\mathrm{pub}(B)}$ but if the encryption is deterministic, an intruder may try different values for $\mathtt{s}$ and check whether the ciphertext he obtained using B's public key is equal to the one he receives. Thus for our result to hold, we require that encryption is probabilistic. This is not a restriction since this is *de facto* the standard in almost all cryptographic applications. Next, we consider the more challenging case of active adversaries. We give sufficient conditions on the protocols for syn-

tactic secrecy to imply strong secrecy (Section 3). Intuitively, we require that the conditional tests are not performed directly on the secret since we have seen above that such tests provide information on the value of this secret. We again exhibit several counter-examples to motivate the introduction of our conditions. An important aspect of our result is that we do not make any assumption on the number of sessions: we put no restriction on the use of replication.

The interest of our contribution is twofold. First, conceptually, it helps to understand when the two definitions of secrecy are actually equivalent. Second, we can transfer many existing results (and the armada of automatic tools) developed for syntactic secrecy. For instance, since the syntactic secrecy problem is decidable for tagged protocols for an unbounded number of sessions [15]. By translating the tagging assumption to the applied-pi calculus, we can derive a first decidability result for strong secrecy for an unbounded number of sessions. Other decidable fragments might be derived from [12] for bounded messages (and nonces) and [3] for a bounded number of sessions.

2 Passive case

2.1 Syntax

Cryptographic primitives are represented by functional symbols. More specifically, we consider the signature $\Sigma = \{\text{enc, dec, enca, deca, } \langle\rangle, \pi_1, \pi_2, \text{ sign, check, pub, priv}\}$. $\mathcal{T}(\Sigma, \mathcal{X}, \mathcal{N})$ denotes the set of terms built over Σ extended by a set of constants, the infinite set of *names* $\mathcal{N}$ and the infinite set of variables $\mathcal{X}$. A term is *closed* or *ground* if it does not contain any variable. The set of names occurring in a term m is denoted by $\text{fn}(m)$, the set of variables is denoted by $\mathcal{V}(m)$. The *positions* in a term t are defined recursively as usual (*i.e.* as sequences of positive integers), ϵ being the empty sequence. $\text{Pos}(t)$ will denote the set of positions of t and $\text{Pos}_{\mathsf{v}}(t)$ the set of positions of variables in t. We denote by $t|_p$ the subterm of t at position p, by $u[v]_p$ the term obtained by replacing in u the subterm at position p by v. For a term u, we denote by h_u the function symbol, name or variable at position ϵ in u. We denote by $\leq_{st}$ (resp.$<_{st}$) the subterm (resp. strict) order. We may simply say that a term v *is* in a term u if v is a subterm of u. If $p = i_1.\cdots.i_n$, where $n \geq 1$, is a position then $\text{pr}(p) = i_1.\cdots.i_{n-1}$ is the *parent* position w.r.t. p. Denote by $\mathbb{N}_+^*$ the set of sequences of positive integers.

We equip the signature with an equational theory E:

$$\begin{cases} \pi_1(\langle z_1, z_2\rangle) = z_1 \\ \pi_2(\langle z_1, z_2\rangle) = z_2 \\ \text{dec}(\text{enc}(z_1, z_2, z_3), z_2) = z_1 \\ \text{deca}(\text{enca}(z_1, \text{pub}(z_2), z_3), \text{priv}(z_2)) = z_1 \\ \text{check}(z_1, \text{sign}(z_1, \text{priv}(z_2)), \text{pub}(z_2)) = \text{ok} \\ \text{retrieve}(\text{sign}(z_1, z_2)) = z_1 \end{cases}$$

The function symbols $\pi_1, \pi_2, \text{dec}, \text{deca}, \text{check}$ and retrieve are called *destructors*. Let $\mathcal{R}_E$ be the corresponding rewrite system (obtained by orienting the equations

from left to right). $\mathcal{R}_E$ is convergent. The normal form of a term t w.r.t. $\mathcal{R}_E$ is denoted by $t{\downarrow}$. Notice that E is also stable by substitution of names. As usual, we write $u \to v$ if there exists θ, a position p in u and $l \to r \in \mathcal{R}_E$ such that $u|_p = l\theta$ and $v = u[r\theta]_p$.

The symbol $\langle \_, \_ \rangle$ represents the pairing function and π_1 and π_2 are the associated projection functions. The term $\mathrm{enc}(m, k, r)$ represents the message m encrypted with the key k. The third argument r reflects that the encryption is probabilistic: two encryptions of the same messages under the same keys are different. The symbol dec stands for decryption. The symbols enca and deca are very similar but in an asymmetric setting, where $\mathrm{pub}(a)$ and $\mathrm{priv}(a)$ represent respectively the public and private keys of an agent a. The term $\mathrm{sign}(m, k)$ represents the signature of message m with key k. check enables to verify the signature and retrieve enables to retrieve the signed message from the signature.[1]

After the execution of a protocol, an attacker knows the messages sent on the network and also in which order they were sent. Such message sequences are organized as *frames* $\varphi = \nu\widetilde{n}.\sigma$, where $\sigma = \{{}^{m_1}/_{y_1}, \ldots, {}^{m_k}/_{y_k}\}$ is a ground substitution and $\widetilde{n}$ is a finite set of names. We denote by $\mathrm{dom}(\varphi) = \mathrm{dom}(\sigma) = \{y_1, \ldots, y_k\}$. The variables y_i enable us to refer to each message. The names in $\widetilde{n}$ are said to be *restricted*. Intuitively, these names are *a priori* unknown to the intruder. A term M is said *public* w.r.t. a frame $\nu\widetilde{n}.\sigma$ (or simply $\widetilde{n}$) if $\mathrm{fn}(M) \cap \widetilde{n} = \emptyset$. The set of restricted names $\widetilde{n}$ might be omitted when it is clear from the context. We usually write νn instead of $\nu\{n\}$, and the same for bigger sets.

2.2 Deducibility

Given a frame φ that represents the history of messages sent during the execution of a protocol, we define the *deduction* relation, denoted by $\varphi \vdash M$. Deducible messages are messages that can be obtained from φ by applying functional symbols and the equational theory E.

$$\frac{}{\nu\tilde{n}.\sigma \vdash x\sigma}\ x \in \mathrm{dom}(\sigma) \qquad \frac{}{\nu\tilde{n}.\sigma \vdash s}\ s \in \mathcal{N}\backslash\widetilde{n}$$

$$\frac{\nu\tilde{n}.\sigma \vdash t_1 \quad \cdots \quad \nu\tilde{n}.\sigma \vdash t_r}{\nu\tilde{n}.\sigma \vdash f(t_1, \ldots, t_r)} \qquad \frac{\nu\tilde{n}.\sigma \vdash t \quad t =_E t'}{\nu\tilde{n}.\sigma \vdash t'}$$

Example 1. The terms k and $\langle k, k' \rangle$ are deducible from the frame $\nu k, k', r.\{{}^{\mathrm{enc}(k,k',r)}/_x, {}^{k'}/_y\}$.

A message is usually said secret if it is not deducible. By opposition to our next notion of secrecy, we say that a term M is *syntactically secret* in φ if $\varphi \nvdash M$.

[1] Signature schemes may disclose partial information on the signed message. To enforce the intruder capabilities, we assume that messages can always be retrieved out of the signature.

2.3 Static equivalence

Deducibility does not always suffice to express the abilities of an intruder.

Example 2. The set of deducible messages is the same for the frames $\varphi_1 = \nu k,n_0,n_1,r_0.\{^{\mathrm{enc}(n_0,k,r_0)}/_x, {}^{\langle n_0,n_1\rangle}/_y, {}^{k}/_z\}$ and $\varphi_2 = \nu k,n_0,n_1,r_1.\{^{\mathrm{enc}(n_1,k,r_1)}/_x, {}^{\langle n_0,n_1\rangle}/_y, {}^{k}/_z\}$, while an attacker is able to detect that the last message corresponds to distinct nonces. In particular, the attacker is able to distinguish the two "worlds" represented by φ_1 and φ_2.

We say that a frame $\varphi = \nu\widetilde{n}.\sigma$ *passes the test* (M, N) where M, N are two terms, denoted by $(M = N)\varphi$, if there exists a renaming of the restricted names in φ such that $(\mathrm{fn}(M) \cup \mathrm{fn}(N)) \cap \widetilde{n} = \emptyset$ and $M\sigma =_E N\sigma$. Two frames $\varphi = \nu\widetilde{n}.\sigma$ and $\varphi' = \nu\widetilde{m}.\sigma'$ are *statically equivalent*, written $\varphi \approx \varphi'$, if they pass the same tests, that is $\mathrm{dom}(\varphi) = \mathrm{dom}(\varphi')$ and for all terms M, N such that $(\mathcal{V}(M) \cup \mathcal{V}(N)) \subseteq \mathrm{dom}(\varphi)$ and $(\mathrm{fn}(M) \cup \mathrm{fn}(N)) \cap (\widetilde{n} \cup \widetilde{m}) = \emptyset$, we have $(M = N)\varphi$ iff $(M = N)\varphi'$.

Example 3. The frames φ_1 and φ_2 defined in Example 2 are not statically equivalent since $(\mathrm{dec}(x, z) = \pi_1(y))\varphi_1$ but $(\mathrm{dec}(x, z) \neq \pi_1(y))\varphi_2$.

Let $\mathtt{s}$ be a free name of a frame $\varphi = \nu\widetilde{n}.\sigma$. We say that $\mathtt{s}$ is *strongly secret* in φ if for every closed public terms M, M' w.r.t. φ, we have $\varphi({}^{M}/_{\mathtt{s}}) \approx \varphi({}^{M'}/_{\mathtt{s}})$ that is, the intruder cannot distinguish the frame instantiated by two terms of its choice. For simplicity we may omit $\mathtt{s}$ and write $\varphi(M)$ instead of $\varphi({}^{M}/_{\mathtt{s}})$.

2.4 Syntactic secrecy implies strong secrecy

Syntactic secrecy is usually weaker than strong secrecy! We first exhibit some examples of frames that preserves syntactic secrecy but not strong secrecy. They all rely on different properties.

Probabilistic encryption. The frame $\psi_1 = \nu\mathtt{s}, k, r.\{^{\mathrm{enc}(\mathtt{s},k,r)}/_x, {}^{\mathrm{enc}(n,k,r)}/_y\}$ does not preserve the strong secrecy of $\mathtt{s}$. Indeed, $\psi_1(n) \not\approx \psi_1(n')$ since $(x = y)\psi_1(n)$ but $(x \neq y)\psi_1(n')$. This would not happen if each encryption used a distinct randomness, that is, if the encryption was probabilistic.

Key position. The frame $\psi_2 = \nu\mathtt{s}, n.\{^{\mathrm{enc}(\langle n,n'\rangle,\mathtt{s},r)}/_x\}$ does not preserve the strong secrecy of $\mathtt{s}$. Indeed, $\psi_2(k) \not\approx \psi_2(k')$ since $(\pi_2(\mathrm{dec}(x, k)) = n')\psi_2(k)$ but $(\pi_2(\mathrm{dec}(x, k)) \neq n)\psi_2(k')$. If $\mathtt{s}$ occurs in key position in some ciphertext, the intruder may try to decrypt the ciphertext since $\mathtt{s}$ is replaced by public terms and check for some redundancy. It may occur that the encrypted message does not contain any verifiable part. In that case, the frame may preserve strong secrecy. It is for example the case of the frame $\nu n\{^{\mathrm{enc}(n,\mathtt{s},r)}/_x\}$. Such cases are however quite rare in practice.

No destructors. The frame $\psi_3 = \nu\mathtt{s}.\{^{\pi_1(\mathtt{s})}/_x\}$ does not preserve the strong secrecy of $\mathtt{s}$ simply because $(x = k)$ is true for $\psi_3(\langle k, k'\rangle)$ while not for $\psi_3(k)$.

Retrieve rule. The retrieve$(\mathrm{sign}(z_1, z_2)) = z_1$ may seem arbitrary since not all signature schemes enable to get the signed message out of a signature. It is actually crucial for our result. For example, the frame $\psi_4 = \nu \mathtt{s}.\{^{\mathrm{sign}(\mathtt{s},\mathrm{priv}(a))}/_x, {}^{\mathrm{pub}(a)}/_y\}$ does not preserve the strong secrecy of $\mathtt{s}$ because $(\mathrm{check}(n, x, y) = \mathrm{ok})$ is true for $\psi_4(n)$ but not for $\psi_4(n')$.

In these four cases, the frames preserve the syntactic secrecy of $\mathtt{s}$, that is $\psi_i \not\vdash \mathtt{s}$, for $1 \leq i \leq 4$.

This leads us to the following definition.

Definition 1. *A frame $\varphi = \nu\widetilde{n}.\sigma$ is* well-formed *w.r.t some name $\mathtt{s}$ if*

1. *Encryption is probabilistic, i.e. for any subterm $\mathrm{enc}(m, t, r)$ of ϕ, for any term $t' \in \phi$ and position p such that $t'|_p = r$ we have $p = q.3$ for some q and $t'|_q = \mathrm{enc}(m, t, r)$. The same condition holds for asymmetric encryption. In addition, if $\mathtt{s}$ occurs in m at a position p' such that no encryption appears along the path from the root to p' then r must be restricted, that is $r \in \widetilde{n}$.*
2. *$\mathtt{s}$ is not part of a key, i.e. for all $\mathrm{enc}(m, t, r)$, $\mathrm{enca}(m', t', r')$, $\mathrm{sign}(u, v)$, $\mathrm{pub}(w)$, $\mathrm{priv}(w')$ subterms of φ, $\mathtt{s} \notin \mathrm{fn}(t, t', v, w, w', n, n')$.*
3. *φ does not contain destructor symbols.*

Condition 1 requires that each innermost encryption above $\mathtt{s}$ contains a restricted randomness. This is not a restriction since $\mathtt{s}$ is meant to be a secret value and such encryptions have to be produced by honest agents and thus contain a restricted randomness.

For well-formed frames, syntactic secrecy is actually equivalent to strong secrecy.

Theorem 1. *Let $\varphi = \nu\widetilde{n}.\sigma$ be a well-formed frame w.r.t $\mathtt{s} \in \widetilde{n}$.*

$$\varphi \not\vdash \mathtt{s} \text{ iff } \nu\widetilde{n}\backslash\{\mathtt{s}\}.\sigma({}^{M}/_{\mathtt{s}}) \approx \nu\widetilde{n}\backslash\{\mathtt{s}\}.\sigma({}^{M'}/_{\mathtt{s}})$$

for all M, M' closed public terms w.r.t. φ.

Proof. We present the skeleton of the proof; all details can be found in Appendix A. Let $\varphi = \nu\widetilde{n}.\sigma$ be a well-formed frame w.r.t. $\mathtt{s} \in \widetilde{n}$. If $\varphi \vdash \mathtt{s}$, this trivially implies that $\mathtt{s}$ is not strongly secret. Indeed, there exists a public term M w.r.t. φ such that $M\sigma =_E \mathtt{s}$ (this can be easily shown by induction on the deduction system). Let n_1, n_2 be fresh names such that $n_1, n_2 \notin \widetilde{n}$ and $n_1, n_2 \notin \mathrm{fn}(\varphi)$. Since $M\sigma({}^{n_1}/_{\mathtt{s}}) =_E n_1$ the frames $\nu\widetilde{n}\backslash\{\mathtt{s}\}.\sigma({}^{n_1}/_{\mathtt{s}})$ and $\nu\widetilde{n}\backslash\{\mathtt{s}\}.\sigma({}^{n_2}/_{\mathtt{s}})$ are distinguishable with the test $(M = n_1)$.

We assume now that $\varphi \not\vdash \mathtt{s}$. We first show that any syntactic equality satisfied by the frame $\varphi({}^{M}/_{\mathtt{s}})$ is already satisfied by φ.

Lemma 1. *Let $\varphi = \nu\widetilde{n}.\sigma$ be a well-formed frame w.r.t. $\mathtt{s} \in \widetilde{n}$, u, v terms such that $\mathcal{V}(u), \mathcal{V}(v) \subseteq \mathrm{dom}(\varphi)$ and M a closed term, u, v and M public w.r.t. $\widetilde{n}$. If $\varphi \not\vdash \mathtt{s}$, $u\sigma({}^{M}/_{\mathtt{s}}) = v\sigma({}^{M}/_{\mathtt{s}})$ implies $u\sigma = v\sigma$. Let t be a subterm of a term in σ that does not contain $\mathtt{s}$. If $\varphi \not\vdash \mathtt{s}$, $t = v\sigma({}^{M}/_{\mathtt{s}})$ implies $t = v\sigma$.*

The key lemma is that any reduction that applies to a deducible term t where $\mathtt{s}$ is replaced by some M, directly applies to t.

Lemma 2. *Let $\varphi = \nu\widetilde{n}.\sigma$ be a well-formed frame w.r.t. $\mathtt{s} \in \widetilde{n}$ such that $\varphi \not\vdash \mathtt{s}$. Let u be a term with $\mathcal{V}(u) \subseteq \mathrm{dom}(\varphi)$ and M be a closed term in normal form, u and M public w.r.t. $\widetilde{n}$. If $u\sigma({}^{M}/_{\mathtt{s}}) \to v$, for some term v , then there exists a well-formed frame $\varphi' = \nu\widetilde{n}.\sigma'$ w.r.t. $\mathtt{s}$*

- *extending φ, that is $x\sigma' = x\sigma$ for all $x \in \mathrm{dom}(\sigma)$,*
- *preserving deducible terms: $\varphi \vdash w$ iff $\varphi' \vdash w$,*
- *and such that $v = v'\sigma'({}^{M}/_{\mathtt{s}})$ for some v' public w.r.t. $\widetilde{n}$.*

This lemma allows us to conclude the proof of Theorem 1. Fix arbitrarily two public closed terms M, M'. We can assume w.l.o.g. that M and M' are in normal form. Let $u \neq v$ be two public terms such that $\mathcal{V}(u), \mathcal{V}(v) \subseteq \mathrm{dom}(\varphi)$ and $u\sigma({}^{M}/_{\mathtt{s}}) =_E v\sigma({}^{M}/_{\mathtt{s}})$. Then there are $u_1, \ldots, u_k$ and $v_1, \ldots, v_l$ such that $u\sigma({}^{M}/_{\mathtt{s}}) \to u_1 \to \ldots \to u_k$, $v\sigma({}^{M}/_{\mathtt{s}}) \to v_1 \to \ldots \to v_l$, $u_k = u\sigma({}^{M}/_{\mathtt{s}})\downarrow$, $v_l = v\sigma({}^{M}/_{\mathtt{s}})\downarrow$ and $u_k = v_l$.

Applying repeatedly Lemma 2 we obtain that there exist public terms $u'_1, \ldots, u'_k$ and $v'_1, \ldots, v'_l$ and well-formed frames $\varphi^{u_i} = \nu\widetilde{n}.\sigma^{u_i}$, for $i \in \{1, \ldots, k\}$ and $\varphi^{v_j} = \nu\widetilde{n}.\sigma^{v_j}$, for $j \in \{1, \ldots, l\}$ (as in the lemma) such that $u_i = u'_i\sigma^{u_i}$ and $v_j = v'_j\sigma^{v_j}$.

We consider $\varphi' = \nu\widetilde{n}.\sigma'$ where $\sigma' = \sigma^{u_k} \cup \sigma^{v_l}$. Since only subterms of φ have been added to φ', it is easy to verify that φ' is still a well-formed frame and for every term w, $\varphi \vdash w$ iff $\varphi' \vdash w$. In particular $\varphi' \not\vdash \mathtt{s}$.

By construction we have that $u'_k\sigma^{u_k}({}^{M}/_{\mathtt{s}}) = v'_l\sigma^{v_l}({}^{M}/_{\mathtt{s}})$. Then, by Lemma 1, we deduce that $u'_k\sigma^{u_k}(\mathtt{s}) = v'_l\sigma^{v_l}(\mathtt{s})$ that it $u\sigma =_E v\sigma$. By stability of substitution of names, we have $u\sigma({}^{M'}/_{\mathtt{s}}) =_E v\sigma({}^{M'}/_{\mathtt{s}})$. We deduce that $\nu\widetilde{n}\backslash\{\mathtt{s}\}.\sigma({}^{M}/_{\mathtt{s}}) \approx \nu\widetilde{n}\backslash\{\mathtt{s}\}.\sigma({}^{M'}/_{\mathtt{s}})$.

3 Active case

To simplify the analyze of the active case, we restrict our attention to pairing and symmetric encryption: the alphabet Σ is now reduced to $\Sigma = \{\mathrm{enc}, \mathrm{dec}, \langle\rangle, \pi_1, \pi_2\}$ and E is limited to the first three equations.

3.1 Modeling protocols within the applied pi calculus

The applied pi calculus [1] is a process algebra well-suited for modeling cryptographic protocols, generalizing the spi-calculus [2]. We shortly describe its syntax and semantics. This part is mostly borrowed from [1].

Processes, also called plain processes, are defined by the grammar:

$P, Q, R :=$ processes

$\mathbf{0}$	null process	$P \mid Q$	parallel composition
$!P$	replication	$\nu n.P$	name restriction
$if\ M = N\ then\ P\ else\ Q$	conditional	$c(z).P$	message input
$\overline{c}\langle M\rangle.P$	message output		

where n is a name, M, N are terms, and c is a name or a variable. The null process $\mathbf{0}$ does nothing. Parallel composition executes the two processes concurrently. Replication $!P$ creates unboundedly new instances of P. Name restriction $\nu n.P$ builds a new, private name n, binds it in P and then executes P. The conditional $if\ M = N\ then\ P\ else\ Q$ behaves like P or Q depending on the result of the test $M = N$. If Q is the null process then we use the notation $[M = N].P$ instead. Finally, the process $c(z).P$ inputs a message and executes P binding the variable z to the received message, while the process $\overline{c}\langle M\rangle.P$ outputs the message M and then behaves like P. We may omit P if it is $\mathbf{0}$. In what follows, we restrict our attention to the case where c is name since it is usually sufficient to model cryptographic protocols.

Extended processes are defined by the grammar:

$A, B :=$ extended processes

P	plain process	$A \mid B$	parallel composition
$\nu n.A$	name restriction	$\nu x.A$	variable restriction
$\{^{M}/_{x}\}$	active substitution		

Active substitutions generalize *let*, in the sense that $\nu x.(\{^{M}/_{x}\}|P)$ corresponds to $let\ x = M\ in\ P$, while unrestricted, $\{^{M}/_{x}\}$ behaves like a permanent knowledge, permitting to refer globally to M by means of x. We identify variable substitutions $\{^{M_1}/_{x_1}, \ldots, ^{M_k}/_{x_k}\}$, $k \geq 0$ with extended processes $\{^{M_1}/_{x_1}\}| \ldots |\{^{M_k}/_{x_k}\}$. In particular the empty substitution is identified with the null process.

We denote by $\mathrm{fv}(A)$, $\mathrm{bv}(A)$, $\mathrm{fn}(A)$, and $\mathrm{bn}(A)$ the sets of free and bound variables and free and bound names of A, respectively, defined inductively as usual for the pi calculus' constructs and using $\mathrm{fv}(\{^{M}/_{x}\}) = \mathrm{fv}(M) \cup \{x\}$ and $\mathrm{fn}(\{^{M}/_{x}\}) = \mathrm{fn}(M)$ for active substitutions. An extended process is *closed* if it has no free variables except those in the domain of active substitutions.

Extended processes built up from the null process (using the given constructions, that is, parallel composition, restriction and active substitutions) are called *frames*[2]. To every extended process A we associate the frame $\varphi(A)$ obtained by replacing all embedded plain processes with $\mathbf{0}$.

An *evaluation context* is an extended process with a hole not under a replication, a conditional, an input or an output.

Structural equivalence ($\equiv$) is the smallest equivalence relation on extended processes that is closed by α-conversion of names and variables, by application of evaluation contexts and such that the standard structural rules for the null process, parallel composition and restriction (such as associativity and commutativity of $|$, commutativity and binding-operator-like behavior of ν) together with the following ones hold.

$$\nu x.\{^{M}/_{x}\} \equiv \mathbf{0} \qquad \text{ALIAS}$$
$$\{^{M}/_{x}\} \mid A \equiv \{^{M}/_{x}\} \mid A\{^{M}/_{x}\} \qquad \text{SUBST}$$
$$\{^{M}/_{x}\} \equiv \{^{N}/_{x}\} \quad \text{if } M =_E N \qquad \text{REWRITE}$$

[2] We see later in this section why we use the same name as for the notion defined in section 2.

If $\widetilde{n}$ represents the (possibly empty) set $\{n_1, \ldots, n_l\}$, we abbreviate by $\nu\widetilde{n}$ the sequence $\nu n_1.\nu n_2 \ldots \nu n_l$. Every closed extended process A can be brought to the form $\nu\widetilde{n}.\{^{M_1}/_{x_1}\}|\ldots|\{^{M_k}/_{x_k}\}|P$ by using structural equivalence, where P is a plain closed process, $k \geq 0$ and $\{\widetilde{n}\} \subseteq \cup_i \mathrm{fn}(M_i)$. Hence the two definitions of frames are equivalent up to structural equivalence on closed extended processes. To see this we apply rule SUBST until all terms are ground (this is assured by the fact that the considered extended processes are closed and the active substitutions are cycle-free). Also, another consequence is that if $A \equiv B$ then $\varphi(A) \equiv \varphi(B)$.

Two semantics can be considered for this calculus, defined by structural equivalence and by *internal reduction* and *labeled reduction*, respectively. These semantics lead to *observational equivalence* (which is standard and not recalled here) and *labeled bisimilarity* relations. The two bisimilarity relations coincide [1] and we use here the latter since it permits to take implicitly into account the observer, hence it has the advantage of relying on static equivalence rather than quantification over contexts.

Internal reduction is the largest relation on extended processes closed by structural equivalence and application of evaluation contexts such that:

$$\overline{c}\langle x\rangle.P \mid c(x).Q \rightarrow P \mid Q \qquad \text{COMM}$$

$$if\ M = M\ then\ P\ else\ Q \rightarrow P \qquad \text{THEN}$$

$$if\ M = N\ then\ P\ else\ Q \rightarrow Q \qquad \text{ELSE}$$

for any ground terms M and N such that $M \neq_E N$

On the other hand, *labeled reduction* is defined by the following rules:

$$c(x).P \xrightarrow{c(M)} P\{^{M}/_{x}\} \quad \text{IN} \qquad \overline{c}\langle u\rangle.P \xrightarrow{\overline{c}\langle u\rangle} P \quad \text{OUT-ATOM}$$

$$\frac{A \xrightarrow{\overline{c}\langle u\rangle} A'}{\nu u.A \xrightarrow{\nu u.\overline{c}\langle u\rangle} A'}\ u \neq c \quad \text{OPEN-ATOM} \qquad \frac{A \xrightarrow{\alpha} A'}{\nu u.A \xrightarrow{\alpha} \nu u.A'}\ \begin{array}{l}u \text{ does not}\\ \text{occur in } \alpha\end{array} \quad \text{SCOPE}$$

$$\frac{A \xrightarrow{\alpha} A'}{A|B \xrightarrow{\alpha} A'|B}\ (*) \quad \text{PAR} \qquad \frac{A \equiv B \quad B \xrightarrow{\alpha} B' \quad B' \equiv A'}{A \xrightarrow{\alpha} A'} \quad \text{STRUCT}$$

where u is a metavariable that ranges over names and variables, and the condition (*) of the rule PAR is $\mathrm{bv}(\alpha) \cap \mathrm{fv}(B) = \mathrm{bn}(\alpha) \cap \mathrm{fn}(B) = \emptyset$.

Definition 2. Labeled bisimilarity *($\approx_l$) is the largest symmetric relation $\mathcal{R}$ on closed extended processes such that $A\,\mathcal{R}\,B$ implies:*

1. *$\varphi(A) \approx \varphi(B)$;*
2. *if $A \rightarrow A'$ then $B \rightarrow^* B'$ and $A'\,\mathcal{R}\,B'$, for some B';*
3. *if $A \xrightarrow{\alpha} A'$ and $\mathrm{fv}(\alpha) \subseteq \mathrm{dom}(\varphi(A))$ and $\mathrm{bn}(\alpha) \cap \mathrm{fn}(B) = \emptyset$ then $B \rightarrow^* \xrightarrow{\alpha} \rightarrow^* B'$ and $A'\,\mathcal{R}\,B'$, for some B'.*

We denote $A \Rightarrow B$ if $A \rightarrow B$ or $A \xrightarrow{\alpha} B$. Also we use the notation $\nu\mathtt{s}\varphi$ for $\nu(\widetilde{n} \cup \{\mathtt{s}\}).\sigma$, where $\varphi = \nu\widetilde{n}.\sigma$.

Definition 3. *A frame φ is* valid *w.r.t. a process P if there is A such that $P \Rightarrow^* A$ and $\varphi \equiv \varphi(A)$.*

Definition 4. *Let P be a closed plain process without variables as channels and $\mathtt{s}$ a free name of P, but not a channel name. We say that $\mathtt{s}$ is* syntactically secret *in P if, for every valid frame φ w.r.t. P, $\mathtt{s}$ is not deducible from $\nu\mathtt{s}\varphi$. We say that $\mathtt{s}$ is* strongly secret *if for any closed terms M, M' such that $\mathrm{bn}(P) \cap (\mathrm{fn}(M) \cup \mathrm{fn}(M')) = \emptyset$, $P(M/\mathtt{s}) \approx_l P(M'/\mathtt{s})$.*

Let $\mathcal{M}_o(P)$ be the set of *outputs* of P, that is the set of terms m such that $\overline{c}\langle m\rangle$ is a message output construct for some channel name c in P, and let $\mathcal{M}_t(P)$ be the set of *operands of tests* of P, where a *test* is a couple $M = N$ occurring in a conditional and its *operands* are M and N. Let $\mathcal{M}(P) = \mathcal{M}_o(P) \cup \mathcal{M}_t(P)$ be the set of *messages* of P. Examples are provided at the end of this section.

The following lemma intuitively states that any message contained in active frame is an output instantiated by messages deduced from previous messages.

Lemma 3. *Let P be a closed plain process, and A be a closed extended process such that $P \Rightarrow^* A$. There are $k \geq 0$, an extended process $B = \nu\widetilde{n}.\sigma_k|P_B$, where P_B is some plain process, and θ a substitution public w.r.t. $\widetilde{n}$ such that: $A \equiv B$, $\{\widetilde{n}\} \subseteq \mathrm{bn}(P)$, for every side of a test or an output M of P_B there is a message M_0 in P (a side of a test or an output respectively), such that $M = M_0\theta\sigma_k$, and, $\sigma_i = \sigma_{i-1} \cup \{^{m_i\theta_i\sigma_{i-1}}/_{y_i}\}$, for all $1 \leq i \leq k$, where m_i is an output in P, θ_i is a substitution public w.r.t. $\widetilde{n}$ and σ_0 is the empty substitution.*

The proof is done by induction on the number of reductions in $P \Rightarrow^* A$. Intuitively, B is obtained by applying the SUBST rule (from left to right) as most as possible until there are no variables left in the plain process. Note that B is unique up to the structural rules different from ALIAS, SUBST and REWRITE. We say that $\varphi(B)$ is the *standard frame* w.r.t. A.

As a running example we consider the Yahalom protocol:

$$\begin{array}{l} A \Rightarrow B : A, N_a \\ B \Rightarrow S : B, \{A, N_a, N_b\}_{K_{bs}} \\ S \Rightarrow A : \{B, K_{ab}, N_a, N_b\}_{K_{as}}, \{A, K_{ab}\}_{K_{bs}} \\ A \Rightarrow B : \{A, K_{ab}\}_{K_{bs}} \end{array}$$

In this protocol, two participants A and B wish to establish a shared key K_{ab}. The key is created by a trusted server S which shares the secret keys K_{as} and K_{bs} with A and B respectively. The protocol is modeled by the following process:

$$P_Y(k_{ab}) = \nu k_{as}, k_{bs}.(!P_A)|(!P_B)|(!\nu k.P_S(k))|P_S(k_{ab})$$

where

$$\begin{array}{l} P_A = \nu n_a.\overline{c}\langle a, n_a\rangle.c(z_a).[b = u_b].[n_a = u_{n_a}].\overline{c}\langle\pi_2(z_a)\rangle \\ P_B = c(z_b).\nu n_b, r_b.\overline{c}\langle b, \mathrm{enc}(\langle\pi_1(z_b), \langle\pi_2(z_b), n_b\rangle\rangle, k_{bs}, r_b)\rangle.c(z'_b).[a = \pi_1(\mathrm{dec}(z'_b, k_{bs}))] \\ P_S(x) = c(z_s).\nu r_s, r'_s.\overline{c}\langle\mathrm{enc}(\langle\pi_1(z_s), \langle x, v_n\rangle\rangle, k_{as}, r_s), \mathrm{enc}(\langle v_a, x\rangle, k_{bs}, r'_s)\rangle \end{array}$$

and $u_b = \pi_1(\mathrm{dec}(\pi_1(z_a), k_{as}))$ $\quad u_{n_a} = \pi_1(\pi_2(\pi_2(\mathrm{dec}(\pi_1(z_a), k_{as}))))$
$v_a = \pi_1(\mathrm{dec}(\pi_2(z_s), k_{bs}))$ $\quad v_n = \pi_2(\mathrm{dec}(\pi_2(z_s), k_{bs}))$

For this protocol the set of outputs and operands of tests are respectively:

$$\mathcal{M}_o(P_Y) = \{\langle a, n_a\rangle, z_a, \pi_2(z_a), \langle b, \mathrm{enc}(\langle \pi_1(z_b), \langle \pi_2(z_b), n_b\rangle\rangle, k_{bs}, r_b)\rangle, z'_b, \\ \mathrm{enc}(\langle \pi_1(z_s), \langle x, v_n\rangle\rangle, k_{as}, r_s), \mathrm{enc}(\langle v_a, x\rangle, k_{bs}, r'_s)\}$$

and $\mathcal{M}_t(P_Y) = \{b, u_b, n_a, u_{n_a}, a, \pi_1(\mathrm{dec}(z'_b, k_{bs}))\}$.

3.2 Our hypotheses

In what follows, we assume $\mathtt{s}$ to be the secret. We restrict ourself to processes with ground terms in key position. Indeed, if keys contained variables, they could also contain the secret and lead to the same kind of attacks as in the passive case. For example, let $P_1 = \nu k, r, r'.(\overline{c}\langle \mathrm{enc}(\mathtt{s}, k, r)\rangle \mid c(z).\overline{c}\langle \mathrm{enc}(a, \mathrm{dec}(z, k), r')\rangle)$. The name $\mathtt{s}$ in P_1 is syntactically secret but not strongly secret. Indeed,

$$\begin{aligned} P_1 &\equiv \nu k, r, r'.(\nu z.(\{^{\mathrm{enc}(\mathtt{s},k,r)}/_z\} \mid \overline{c}\langle z\rangle \mid c(z).\overline{c}\langle \mathrm{enc}(a, \mathrm{dec}(z, k), r')\rangle)) \\ &\to \nu k, r, r'.(\{^{\mathrm{enc}(\mathtt{s},k,r)}/_z\} \mid \overline{c}\langle \mathrm{enc}(a, \mathtt{s}, r')\rangle) \qquad \text{(COMM rule)} \\ &\equiv \nu k, r, r'.(\nu z'.(\{^{\mathrm{enc}(\mathtt{s},k,r)}/_z, {}^{\mathrm{enc}(a,\mathtt{s},r')}/_{z'}\} \mid \overline{c}\langle z'\rangle)) \\ &\xrightarrow{\nu z'.\overline{c}\langle z'\rangle} \nu k, r, r'.\{^{\mathrm{enc}(\mathtt{s},k,r)}/_z, {}^{\mathrm{enc}(a,\mathtt{s},r')}/_{z'}\}, \end{aligned}$$

and the resulting frame does not preserve the strong secrecy of $\mathtt{s}$ (see the frame ψ_2 of section 2.4).

Also, as in the passive case, destructors above the secret must be forbidden. Indeed, in $P_2 = \overline{c}\langle \pi_1(\mathtt{s})\rangle \equiv \nu z.(\{^{\pi_1(\mathtt{s})}/_z\} | \overline{c}\langle z\rangle) \xrightarrow{\nu z.\overline{c}\langle z\rangle} \{^{\pi_1(\mathtt{s})}/_z\}$, $\mathtt{s}$ is syntactically secret but not strongly secret (see the frame ψ_3 of Section 2.4).

Without loss of generality with respect to cryptographic protocols, we assume that terms occurring in processes are in normal form and that no destructor appears above constructors. Indeed, terms like $\pi_1(\mathrm{enc}(m, k, r))$ are usually not used to specify protocols. We also assume that tests do not contain constructors. Indeed a test $[\langle M_1, M_2\rangle = N]$ can be rewritten as $[M_1 = N_1].[M_2 = N_2]$ if $N = \langle N_1, N_2\rangle$, and $[M_1 = \pi_1(N)].[M_2 = \pi_2(N)]$ if N does not contain constructors, and will never hold otherwise. Similar rewriting applies for encryption, except for the test $[\mathrm{enc}(M_1, M_2, M_3) = N]$ if N does not contain constructors. It can be rewritten in $[\mathrm{dec}(N, M_2) = M_1]$ but this is not equivalent. However since the randomness of encryption is not known to the agent, explicit tests on the randomness should not occur in general.

This leads us to consider the following class of processes. But first, we say that an occurrence q_{enc} of an encryption in a term t is an *agent encryptions* w.r.t. a set of names $\widetilde{n}$ if $t|_{q_{\mathrm{enc}}} = \mathrm{enc}(u, v, r)$ for some u, v, r and $r \in \widetilde{n}$.

Definition 5. *A process P is* well-formed *w.r.t. a name $\mathtt{s}$ if it is closed and if:*

1. *any occurrence of $\mathrm{enc}(m, k, r)$ in some term $t \in \mathcal{M}$ is an agent encryption w.r.t. $\mathrm{bn}(P)$, and for any term $t' \in \mathcal{M}$ and position p such that $t'|_p = r$ there is a position q such that $q.3 = p$ and $t'|_q = \mathrm{enc}(m, k, r)$;*

2. *for every term* $\mathrm{enc}(m,k,r)$ *or* $\mathrm{dec}(m,k)$ *occurring in* P, k *is ground;*
3. *any left or right side of a test* $M \in \mathcal{M}_t$ *is a name, a constant or has the form* $\pi^1(\mathrm{dec}(\ldots\pi^n(\mathrm{dec}(\pi^{n+1}(z),k_n))\ldots,k_1))$, *with* $n \geq 0$, *where the* π^i *are words on* $\{\pi_1,\pi_2\}$ *and* z *is a variable.*
4. *there are no destructors above constructors, nor above* $\mathtt{s}$.

Conditional tests should not test on $\mathtt{s}$. For example, consider the following process:

$$P_3 = \nu k, r.(\overline{c}\langle \mathrm{enc}(\mathtt{s},k,r)\rangle \mid c(z).[\mathrm{dec}(z,k) = a].\overline{c}\langle \mathrm{ok}\rangle)$$

where a is a non restricted name. $\mathtt{s}$ in P_3 is syntactically secret but not strongly secret. Indeed, $P_3 \to \nu k, r.(\{^{\mathrm{enc}(\mathtt{s},k,r)}/_z\} \mid [\mathtt{s} = a].\overline{c}\langle \mathrm{ok}\rangle)$. The process $P_3(^a/_{\mathtt{s}})$ reduces further while $P_3(^b/_{\mathtt{s}})$ does not.

That is why we have to prevent hidden tests on $\mathtt{s}$. Such tests may occur nested in equality tests. For example, let

$$\begin{aligned} P_4 = \nu k, r, r_1, r_2.(&\overline{c}\langle \mathrm{enc}(\mathtt{s},k,r)\rangle \mid \overline{c}\langle \mathrm{enc}(\mathrm{enc}(a,k',r_2),k,r_1)\rangle \\ &\mid c(z).[\mathrm{dec}(\mathrm{dec}(z,k),k') = a].\overline{c}\langle \mathrm{ok}\rangle) \\ \to P_4' = \nu k, r, r_1, r_2.(&\{^{\mathrm{enc}(\mathtt{s},k,r)}/_z\} \mid \overline{c}\langle \mathrm{enc}(\mathrm{enc}(a,k',r_2),k,r_1)\rangle \mid [\mathrm{dec}(\mathtt{s},k') = a].\overline{c}\langle \mathrm{ok}\rangle) \end{aligned}$$

Then $P_4(^{\mathrm{enc}(a,k',r')}/_{\mathtt{s}})$ is not equivalent to $P_4(^n/_{\mathtt{s}})$, since the process $P_4'(^{\mathrm{enc}(a,k',r')}/_{\mathtt{s}})$ emits the message ok while $P_4'(^n/_{\mathtt{s}})$ does not. This relies on the fact that the decryption $\mathrm{dec}(z,k)$ allows access to $\mathtt{s}$ in the test.

For the rest of the section we assume z is a new fixed variable.

To prevent hidden tests on the secret, we compute an over-approximation of the ciphertexts that may contain the secret, by marking with a symbol $\mathtt{x}$ all positions under which the secret may appear in clear.

We first introduce a function f_{ep} that extracts the least encryption over $\mathtt{s}$ and "clean" the pairing function above $\mathtt{s}$. Formally, we define the partial function

$$f_{ep} \colon \mathcal{T} \times \mathbb{N}_+^* \hookrightarrow \mathcal{T} \times \mathbb{N}_+^*$$

$f_{ep}(u,p) = (v,q)$ where v and q are defined as follows: $q \leq p$ is the position (if it exists) of the lowest encryption on the path p in u. If q does not exist or if p is not a maximal position in u, then $f_{ep}(u,p) = \bot$. Otherwise, v is obtained from $u|_q$ by replacing all arguments of pairs that are not on the path p with new variables. More precisely, q is a sequence of the form $i \cdot i_1 \cdots i_k$. We introduce two functions pair_1 and pair_2 defined as follows: $\mathsf{pair}_1(M,N) = \langle M,N\rangle$ and $\mathsf{pair}_2(M,N) = \langle N,M\rangle$. Let $v' = u|_q$. v' must be of the form $\mathrm{enc}(M_1,M_2,M_3)$ with $M_i = \mathsf{pair}_{i_1}(\ldots(\mathsf{pair}_{i_k}(a,N_{i_k}),\ldots),N_{i_1})$ for some constant or variable a and some terms N_{i_j} (remember that q leads to the lowest encryption on the path p). Then v is defined by $v = \mathrm{enc}(M_1',M_2',M_3')$ with $M_j' = M_j$ for $j \neq i$ and $M_i' = \mathsf{pair}_{i_1}(\ldots(\mathsf{pair}_{i_k}(a,x_k),\ldots),x_1)$, where the x_j are fresh variables.

For example,

$$f_{ep}(\mathrm{enc}(\mathrm{enc}(\langle\langle a,b\rangle,c\rangle,k_2,r_2),k_1,r_1),1.1.2) = (\mathrm{enc}(\langle z_{1.1},c\rangle,k_2,r_2),1).$$

The function f_e is the composition of the first projection with f_{ep}.

With the function f_{ep}, we can extract from the outputs of a protocol P the set of ciphertexts where $\mathtt{s}$ appears in clear below the encryption.

$$\mathcal{E}_0(P) = \{f_e(m[\mathtt{x}]_p, p) \mid m \in \mathcal{M}_o(P) \ \wedge \ m|_p = \mathtt{s}\}.$$

For example, $\mathcal{E}_0(P_Y) = \{\mathrm{enc}(\langle z_1, \langle \mathtt{x}, z_{1.2}\rangle\rangle, k_{as}), \mathrm{enc}(\langle z_1, \mathtt{x}\rangle, k_{bs})\}$, where P_Y is the process corresponding to the Yahalom protocol defined in previous section.

However $\mathtt{s}$ may appear in other ciphertexts later on during the execution of the protocol after decryptions and encryptions. Thus we also extract from outputs the destructor parts (which may open encryptions). Namely, we define the partial function

$$f_{dp} \colon \mathcal{T} \times \mathbb{N}_+^* \hookrightarrow \mathcal{T} \times \mathbb{N}_+^*$$

$f_{dp}(u, p) = (v, q)$ where v and q are defined as follows: $q \leq p$ is the occurrence of the highest destructor above p (if it exists). Let $r \leq p$ be the occurrence of the lowest decryption above p (if it exists). Then $v = (u[\mathrm{z}]_{r.1})|_q$. If q or r do not exist then $f_{dp}(u, p) = \bot$.

For example, $f_{dp}(\mathrm{enc}(\pi_1(\mathrm{dec}(\pi_2(y), k_1)), k_2, r_2), 1.1.1.1) = (\pi_1(\mathrm{dec}(\mathrm{z}, k_1)), 1)$

The function f_d is the composition of the first projection with f_{dp}. By applying the function f_d to messages of a well-formed process P we always obtain terms d of the form $d = d_1(\dots d_n)$ where $d_i = \pi^i(\mathrm{dec}(\mathrm{z}, k_i))$ with $1 \leq i \leq n$, k_i are ground terms and π^i is a (possibly empty) sequence of projections $\pi_{j_1}(\pi_{j_2}(\dots(\pi_{j_l})\dots))$.

With the function f_d, we can extract from the outputs of a protocol P the meaningful destructor part.

$$\mathcal{D}_o(P) = \{f_d(m, p) \mid m \in \mathcal{M}_o(P) \ \wedge \ p \in \mathrm{Pos_v}(m)\}$$

For example, $\mathcal{D}_o(P_Y) = \{\pi_2(\mathrm{dec}(\mathrm{z}, k_{bs})), \pi_1(\mathrm{dec}(\mathrm{z}, k_{bs}))\}$.

We are now ready to mark (with $\mathtt{x}$) all the positions where the secret might be transmitted (thus tested). We also define inductively the sets $\mathcal{E}_i(P)$ as follows. For each element e of $\mathcal{E}_i$ we can show that there is an unique term in normal form denoted by $\overline{e}$ such that $\mathcal{V}(\overline{e}) = \{\mathrm{z}\}$ and $\overline{e}(e)\!\downarrow = \mathtt{x}$. For example, let $e_1 = \mathrm{enc}(\langle z_1, \langle \mathtt{x}, z_2\rangle\rangle, k_{as})$, then $\overline{e_1} = \pi_1(\pi_2(\mathrm{dec}(\mathrm{z}, k_{as})))$. We define

$$\begin{aligned}
\overline{\mathcal{E}_i}(P) &= \{u \mid \exists e \in \mathcal{E}_i(P), u \leq_{st} \overline{e} \ \text{ and } \exists q \in \mathrm{Pos}(u), h_{u|_q} = \mathrm{dec}\} \\
\mathcal{E}_{i+1}(P) &= \{m'[\mathtt{x}]_q \mid \exists m \in \mathcal{M}_o(P), p \in \mathrm{Pos_v}(m) \text{ s.t. } f_{ep}(m, p) = (m', p'), \\
&\qquad\qquad f_{dp}(m', p'') = (d, q), p = p'.p'', \text{ and } d_1 \in \overline{\mathcal{E}}_i(P)\}
\end{aligned}$$

For example,

$$\begin{aligned}
&\overline{\mathcal{E}_0}(P_Y) = \{\pi_1(\pi_2(\mathrm{dec}(\mathrm{z}, k_{as}))), \pi_2(\mathrm{dec}(\mathrm{z}, k_{as})), \mathrm{dec}(\mathrm{z}, k_{as}), \pi_2(\mathrm{dec}(\mathrm{z}, k_{bs})), \mathrm{dec}(\mathrm{z}, k_{bs})\} \\
&\mathcal{E}_1(P_Y) = \{\mathrm{enc}(\langle z_1, \langle z_{1.2}, \mathtt{x}\rangle\rangle, k_{as})\} \\
&\overline{\mathcal{E}_1}(P_Y) = \{\pi_2(\pi_2(\mathrm{dec}(\mathrm{z}, k_{as}))), \pi_2(\mathrm{dec}(\mathrm{z}, k_{as})), \mathrm{dec}(\mathrm{z}, k_{as})\}
\end{aligned}$$

and $\mathcal{E}_i(P_Y) = \emptyset$ for $i \geq 2$.

Fact The set $\mathcal{E}(P) = \cup_{i \geq 0} \mathcal{E}_i(P)$ is finite up-to renaming of the variables.

Proof. For every $i \geq 1$, every term $m \in \mathcal{E}_i(P)$, $\mathrm{Pos}(m)$ is included in the (finite) set of positions occurring in terms of $\mathcal{M}_0$.

We can now define an over-approximation of the set of tests that may be applied over the secret.

$$\mathcal{M}_t^{\mathtt{s}}(P) = \{M \in \mathcal{M}_t(P) \mid p \in \mathrm{Pos_v}(M) \\ \text{and } \; d = f_{dp}(M,p) \neq \bot \quad \text{and } \exists e \in \mathcal{E}, \exists i, \text{ s.t.} \\ d_i = \pi^i(\mathrm{dec(z)}, k), e = \mathrm{enc}(u,k) \text{ and } \mathtt{x} \in d_i(e)\downarrow\}$$

For example, $\mathcal{M}_t^{\mathtt{s}}(P_Y) = \{\pi_1(\pi_2(\pi_2(\mathrm{dec}(\pi_1(z_a), k_{as}))))\}$.

Definition 6. *We say that a well-formed process P w.r.t.* $\mathtt{s}$ does not test over $\mathtt{s}$ *if the following conditions are satisfied:*

1. *for all $e \in \mathcal{E}(P)$, for all $d = d_1(\ldots d_n) \in \mathcal{D}_o(P)$, if $d_i = \pi^i(\mathrm{dec(z)}, k)$ and $e = \mathrm{enc}(u,k)$ and $\mathtt{x} \in d_i(e)\downarrow$ then $i = 1$ and $\overline{e} \not<_{st} d_1$*
2. *if $M = N$ or $N = M$ is a test and $M \in \mathcal{M}_t^{\mathtt{s}}(P)$ then N is a restricted name.*

Note that $\mathcal{E}(P)$ can be computed in polynomial time from P and that whether P *does not test over* $\mathtt{s}$ is decidable. We show in the next section that the first condition is sufficient to ensure that frames obtained from P are well-formed. It ensures in particular that there are no destructors right above $\mathtt{s}$. If some d_i cancels some encryption in some e and $\mathtt{x} \in d_i(e)\downarrow$ then all its destructors should reduce in the normal form computation (otherwise some destructors (namely projections from d_i) remain above $\mathtt{x}$). Also we have $i = 1$ since otherwise a d_i may have consumed the lowest encryption above $\mathtt{x}$, thus the other decryption may block, and again there would be destructors left above $\mathtt{x}$.

The second condition requires that whenever a side of a test $M = N$ is potentially dangerous (that is M or $N \in \mathcal{M}_t^{\mathtt{s}}(P)$) then the other side should be a restricted name.

3.3 Main result

We are now ready to prove that syntactic secrecy is actually equivalent to strong secrecy for protocols that are well-formed and does not test over the secret.

Theorem 2. *Let P be well-formed process w.r.t. a free name* $\mathtt{s}$*, which is not a channel name, such that P does not test over* $\mathtt{s}$*. We have $\nu\mathtt{s}\varphi \vdash \mathtt{s}$ for any valid frame φ w.r.t. P if and only if $P(M/\mathtt{s}) \approx_l P(M'/\mathtt{s})$, for all ground terms M, M' public w.r.t.* $\mathrm{bn}(P)$.

Proof. Again, we only provide a sketch of the proof. Showing that strong secrecy implies syntactic secrecy is simple so we concentrate here on the converse implication. Let P be well-formed process w.r.t. a nonce $\mathtt{s}$ with no test over $\mathtt{s}$ and assume that P is syntactically secret w.r.t. $\mathtt{s}$.

Let M, M' be to public terms w.r.t. $\mathrm{bn}(P)$. To prove that $P(^{M}/_{\mathtt{s}})$ and $P(^{M'}/_{\mathtt{s}})$ are labeled bisimilar, we need to show that each move of $P(^{M}/_{\mathtt{s}})$ can be matched by $P(^{M'}/_{\mathtt{s}})$ such that the corresponding frames are bisimilar (and conversely). By hypothesis, P is syntactically secret w.r.t. $\mathtt{s}$ thus for any valid frame φ w.r.t. P, we have $\nu\mathtt{s}\varphi \not\vdash \mathtt{s}$. In order to apply our previous result in the passive setting (Theorem 1), we need to show that all the valid frames are well-formed. However, frames may now contain destructors in particular if the adversary sends messages that contain destructors. Thus we first need to extend our definition of well-formedness for frames.

Definition 7. *We say that a frame $\varphi = \nu\widetilde{n}.\sigma$ is* extended well-formed *w.r.t.* $\mathtt{s}$ *if for every occurrence $q_{\mathtt{s}}$ of $\mathtt{s}$ in $t{\downarrow}$, where $t = x\sigma$ for some $x \in \mathrm{dom}(\sigma)$, there exists an agent encryption w.r.t. $\widetilde{n}$ above $\mathtt{s}$. Let $q_{\mathrm{enc}} < q_{\mathtt{s}}$ the occurrence of the lowest encryption. It must verify that $\{h_{t|_q} \mid q_{\mathrm{enc}} < q < q_{\mathtt{s}}\} \subseteq \{\langle,\rangle\}$.*

This definition ensures in particular that there is no destructor directly above $\mathtt{s}$.

Theorem 1 can easily be generalized to extended well-formed frames.

Proposition 1. *Let $\varphi = \nu(\widetilde{n} \uplus \{\mathtt{s}\}).\sigma$ be an extended well-formed frame w.r.t $\mathtt{s}$. $\varphi \not\vdash \mathtt{s}$ iff $\nu\widetilde{n}.\sigma(^{M}/_{\mathtt{s}}) \approx \nu\widetilde{n}.\sigma(^{M'}/_{\mathtt{s}})$ for all M, M' closed public terms w.r.t. φ.*

The proof is obtained by adapting the proof of Theorem 1.

The first step of the proof of Theorem 2 is to show that any frame produced by the protocol is a extended well-formed frame. We actually prove directly a stronger result, crucial in the proof: the secret $\mathtt{s}$ always occurs under an honest encryption and this subterm is an instance of a term in $\mathcal{E}$.

Lemma 4. *Let P be a well-formed process with no test over $\mathtt{s}$ and $\varphi = \nu\widetilde{n}.\sigma$ be a valid frame w.r.t. P such that $\nu\mathtt{s}\varphi \not\vdash \mathtt{s}$. Consider the corresponding standard frame $\nu\widetilde{n}.\overline{\sigma} = \nu\widetilde{n}.\{t_j \mid 1 \le j \le k\}$. For every occurrence $q_{\mathtt{s}}$ of $\mathtt{s}$ in $t_j{\downarrow}$, we have $f_e(t_j{\downarrow}, q_{\mathtt{s}}) = e[^{w}/_{\mathtt{x}}]$ for some $e \in \mathcal{E}$ and some term w. In addition $\nu\widetilde{n}.\sigma_j{\downarrow}$ is an extended well-formed frame w.r.t. $\mathtt{s}$.*

The lemma is proved by induction on j and relies deeply on the construction of the $\mathcal{E}_l$.

The second step of the proof consists in showing that any successful test in the process $P(^{M}/_{\mathtt{s}})$ is also successful in P thus in $P(^{M'}/_{\mathtt{s}})$.

Lemma 5. *Let P be a well-formed process with no test over $\mathtt{s}$, $\varphi = \nu\widetilde{n}.\sigma$ a valid frame for P such that $\nu\mathtt{s}\varphi \not\vdash \mathtt{s}$ and θ a public substitution. If $T_1 = T_2$ is a test in P, then $T_1\theta\sigma(^{M}/_{\mathtt{s}}) =_E T_2\theta\sigma(^{M}/_{\mathtt{s}})$ implies $T_1\theta\sigma =_E T_2\theta\sigma$.*

This lemma is proved by case analysis, depending on whether $T_1, T_2 \in \mathcal{M}_t^{\mathtt{s}}$ and whether $\mathtt{s}$ occurs or not in $\mathrm{fn}(T_1\theta\sigma)$ and $\mathrm{fn}(T_2\theta\sigma)$.

To prove that $P(^{M}/_{\mathtt{s}})$ and $P(^{M'}/_{\mathtt{s}})$ are labeled bisimilar, we introduce the following relation $\mathcal{R}$ between extended processes defined as follows: $A\,\mathcal{R}\,B$ if there is an extended process A_0 and terms M, M' such that $P \Rightarrow^* A_0$, $A = A_0(^{M}/_{\mathtt{s}})$ and $B = A_0(^{M'}/_{\mathtt{s}})$.

Then we show that $\mathcal{R}$ satisfies the three points of the definition of labeled bisimilarity using in particular Lemma 5. Hence we have also $\mathcal{R} \subseteq \approx_l$. Since we have clearly that $P(^{M}/_{\mathtt{s}})\,\mathcal{R}\,P(^{M'}/_{\mathtt{s}})$, it follows that $P(^{M}/_{\mathtt{s}}) \approx_l P(^{M'}/_{\mathtt{s}})$.

3.4 Examples

We have seen in Section 3.2 that P_Y is a well-formed process w.r.t. k_{ab} and does not test over k_{ab}. Applying Theorem 2, if P_Y preserves the syntactic secrecy of k_{ab}, we can deduce that the Yahalom protocol preserves the strong secrecy of k_{ab} that is

$$P_Y({}^{M}/_{k_{ab}}) \approx_l P_Y({}^{M'}/_{k_{ab}})$$

for any public terms M, M' w.r.t. $\mathrm{bn}(P_Y)$. We did not formally prove that the Yahalom protocol preserves the syntactic secrecy of k_{ab} but this was done with several tools in slightly different settings (e.g.[8, 14]).

We have also verified that the Needham-Schroeder symmetric key protocol and the Wide-Mouthed-Frog protocol are both well-formed process w.r.t. k_{ab} and do not test over k_{ab}, where k_{ab} is the exchanged key. Again, the syntactic secrecy of k_{ab} has been proved by several tools (e.g. [8]) in slightly different settings for both protocols. Using Theorem 2, we can deduce that they both preserve the strong secrecy of k_{ab}.

4 Conclusion

We have shown how syntactic secrecy actually implies strong secrecy in both passive and active setting under some conditions, motivated by counterexamples.

We plan to further investigate the active case by considering in particular other primitives like asymmetric encryption and signatures and trying to relax our conditions for specific classes of protocols such as ping-pong protocols. We hope to derive in that way new decidability results for strong secrecy, based on the known ones for syntactic secrecy.

References

1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115. ACM Press, January 2001.
2. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *4th ACM Conference on Computer and Communications Security (CCS'97)*, pages 36–47. ACM Press, 1997.
3. R. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. In *12th International Conference on Concurrency Theory (CONCUR'00)*, volume 1877 of *LNCS*, pages 380–394, 2000.
4. The AVISPA Project. http://www.avispa-project.org/.
5. B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Computer Security Foundations Workshop (CSFW'01)*, pages 82–96. IEEE Comp. Soc. Press, 2001.
6. B. Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *IEEE Symposium on Security and Privacy (SP'04)*, pages 86–100, 2004.
7. B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. In *Foundations of Software Science and Computation Structures (FoSSaCS'03)*, volume 2620 of *LNCS*, April 2003.

8. L. Bozga, Y. Lakhnech, and M. Périn. HERMES: An automatic tool for verification of secrecy in security protocols. In *15th Int. Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 219–222. Springer, 2003.
9. H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In *International Conference on Rewriting Techniques and Applications (RTA'2003)*, LNCS 2706, pages 148–164. Springer-Verlag, 2003.
10. V. Cortier and B. Warinschi. Computationally Sound, Automated Proofs for Security Protocols. In *European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 157–171. Springer, April 2005.
11. G. Denker, J. Millen, and H. Rueß. The CAPSL Integrated Protocol Environment. Technical Report SRI-CSL-2000-02, SRI International, Menlo Park, CA, 2000.
12. N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols*, 1999.
13. H. Hüttel. Deciding framed bisimilarity. In *INFINITY'02*, August 2002.
14. L. C. Paulson. Relations between secrets: Two formal analyses of the Yahalom protocol. *Journal of Computer Security*, 9(3):197–216, 2001.
15. R. Ramanujam and S.P.Suresh. Tagging makes secrecy decidable for unbounded nonces as well. In *23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03)*, Mumbai, 2003.
16. M. Rusinowitch and M. Turuani. Protocol Insecurity with Finite Number of Sessions and Composed Keys is NP-complete. *Theoretical Computer Science*, 299:451–475, 2003.
17. K. N. Verma, H. Seidl, and Th. Schwentick. On the complexity of equational horn clauses. In *22th International Conference on Automated Deduction (CADE 2005)*, LNCS, pages 337–352. Springer-Verlag, 2005.

A Passive case

We prove here Lemmas 1 and 2 of Section 2.

We define $\mathrm{Pos}_{\mathrm{nv}}(u) = \{p \in \mathrm{Pos}(u) \mid u|_p \notin \mathcal{V}(u)\}$ to be the set of non-variable positions of u. We also define the partial function $\mathrm{sf} : \mathbb{N}_+^* \times \mathbb{N}_+^* \hookrightarrow \mathbb{N}_+^*$, $\mathrm{sf}(p,q) = r$ if $p = q.r$ and $\mathrm{sf}(p,q) = \perp$ otherwise.

We first start by an initial lemma that states that in a well-formed frame w.r.t. $\mathtt{s}$, either every occurrence of $\mathtt{s}$ is under some encryption or $\mathtt{s}$ is deducible.

Lemma 6. *Let $\varphi = \nu\widetilde{n}.\sigma$ be a well-formed frame w.r.t. $\mathtt{s} \in \widetilde{n}$ and let p be an occurrence of $\mathtt{s}$ in $y\sigma(\mathtt{s})$ for some $y \in \mathrm{dom}(\sigma)$. If $\varphi \nvdash \mathtt{s}$ then there exists a position $q < p$ such that $y\sigma(\mathtt{s})|_q$ is an encryption, that is $h_{y\sigma(\mathtt{s})|_q} \in \{\mathrm{enc}, \mathrm{enca}\}$; In addition, $\mathtt{s}$ occur in the plaintext subterm of the encrypted term, that is $q \cdot 1 \leq p$.*

Proof. Assume by contradiction that there is an occurrence of $\mathtt{s}$ such that there is no encryption above $\mathtt{s}$. Then, from Properties 2 and 3 of well-formed frames, we have that there are only pairs and signatures as function symbols above $\mathtt{s}$. Hence $\mathtt{s}$ is deducible. Thus there exists a position $q < p$ such that $y\sigma(\mathtt{s})|_q$ is an encryption. By property 2 of well-formed frames, $\mathtt{s}$ must occur in the plain-text part of the encryption that is $q \cdot 1 \leq p$.

We are now ready to prove Lemma 1.

Lemma 1. *Let $\varphi = \nu\widetilde{n}.\sigma$ be a well-formed frame w.r.t. $\mathtt{s} \in \widetilde{n}$, u, v terms such that $\mathcal{V}(u), \mathcal{V}(v) \subseteq \mathrm{dom}(\varphi)$ and M a closed term, u, v and M public w.r.t. $\widetilde{n}$. If $\varphi \nvdash \mathtt{s}$, $u\sigma({}^{M}/_{\mathtt{s}}) = v\sigma({}^{M}/_{\mathtt{s}})$ implies $u\sigma = v\sigma$. Let t be a subterm of a term in σ that does not contain $\mathtt{s}$. If $\varphi \nvdash \mathtt{s}$, $t = v\sigma({}^{M}/_{\mathtt{s}})$ implies $t = v\sigma$.*

Proof. Suppose that $u\sigma({}^{M}/_{\mathtt{s}}) = v\sigma({}^{M}/_{\mathtt{s}})$ and $u\sigma(\mathtt{s}) \neq v\sigma(\mathtt{s})$. Then there is an occurrence p of $\mathtt{s}$, say in $u\sigma$, such that $v\sigma|_p \neq \mathtt{s}$. Consider the variable $y \in \mathcal{V}(u) \subseteq \mathrm{dom}(\sigma)$ and its occurrence p_y in u such that $p = p_y \cdot p'$ for some p'.

By Lemma 6, there is an encryption position q in $y\sigma(\mathtt{s})$ such that $q \cdot 1 \leq p'$. We assume q to be the innermost encryption above $\mathtt{s}$, that is q is maximal. Hence by Property 1 of well-formed frames, the term at position $q \cdot 3$ is a restricted name. It results that $p_y \cdot q \cdot 3 \notin \mathrm{Pos}_{\mathrm{nv}}(v)$, since v is public. Thus there is a variable $y' \in \mathcal{V}(v) \subseteq \mathrm{dom}(\sigma)$ at position $p_{y'}$ in v such that $p_{y'} \leq p_y \cdot q \cdot 3$. Let $m = y\sigma(\mathtt{s})$ and $m' = y'\sigma(\mathtt{s})$. Let q' such that $p_y \cdot q = p_{y'} \cdot q'$. Since $m|_{q \cdot 3} = m'|_{q' \cdot 3}$, we have, by the properties of probabilistic encryptions, that $m|_q = m'|_{q'}$. Since $p_y \cdot q = p_{y'} \cdot q'$ this means in particular that $u\sigma|_p = v\sigma|_p = \mathtt{s}$, which contradicts the fact that $v\sigma(\mathtt{s})|_p \neq \mathtt{s}$.

Let t be a subterm of a term in σ that does not contain $\mathtt{s}$. The proof that $t = v\sigma({}^{M}/_{\mathtt{s}})$ implies $t = v\sigma$ is done similarly.

We now prove key Lemma 2 of Section 2.

Lemma 2. *Let $\varphi = \nu\widetilde{n}.\sigma$ be a well-formed frame w.r.t. $\mathtt{s} \in \widetilde{n}$ such that $\varphi \nvdash \mathtt{s}$. Let u be a term with $\mathcal{V}(u) \subseteq \mathrm{dom}(\varphi)$ and M be a closed term in normal form, u and M public w.r.t. $\widetilde{n}$. If $u\sigma({}^{M}/_{\mathtt{s}}) \to v$, for some term v , then there exists a well-formed frame $\varphi' = \nu\widetilde{n}.\sigma'$ w.r.t. $\mathtt{s}$*

– *extending* φ*, that is* $x\sigma' = x\sigma$ *for all* $x \in \mathrm{dom}(\sigma)$*,*
– *preserving deducible terms:* $\varphi \vdash w$ *iff* $\varphi' \vdash w$*,*
– *and such that* $v = v'\sigma'(^{M}/_{\mathtt{s}})$ *for some* v' *public w.r.t.* $\widetilde{n}$*.*

Proof. Let u, v, M be public terms, M being closed and in normal form such that $u\sigma(^{M}/_{\mathtt{s}}) \to v$, as in the statement of the lemma. Let $l \to r \in \mathcal{R}_E$ be the rule that was applied in the above reduction and let p be the position at which it was applied, *i.e.* $u\sigma(^{M}/_{\mathtt{s}})|_p = l\theta$.

This position p must be in $u\sigma$ since M is in normal form. In addition, since the head function symbol of l is a destructor, by Condition 3 of well-formed frames, p must be in u.

So let $t = u|_p$. We have $t\sigma(^{M}/_{\mathtt{s}}) = l\theta$.

Assume that there is a substitution θ_0 such that $t\sigma = l\theta_0$. This will be proved in Claim 1 below.

For our equational theory E, r is either a constant or a variable. If r is a constant then we take $v' = u[r]_p$ and $\sigma' = \sigma$. It is easy to verify that the conditions of Lemma 2 are satisfied in this case.

Suppose now that r is a variable z_0. Then, consider the[3] unique position q of z_0 in l. This position q is also in $l\theta_0$, that is, in $t\sigma$. So we can have that q is a position in t, but not in $t\sigma$, or, that q is a position in $t\sigma$, but not in t (or $t|_p$ is a variable). Hence we can have:

1. If q is a position in t, but not in $t\sigma$ (that is, there is no $y \in \mathrm{dom}(\varphi)$ above z_0) then we consider $v' = u[t|_q]_p$ and $\sigma' = \sigma$. In this case also, it is easy to verify that the conditions of the Lemma 2 are satisfied.
2. If q is a position in $t\sigma$, but not in t (that is, there is some $y \in \mathrm{dom}(\varphi)$ above z_0). Then we consider $v' = u[y']_p$ and $\sigma' = \sigma \cup \{r\theta_0/y'\}$, where $y' \notin \mathrm{dom}(\sigma)$. We have that $t\sigma =_E r\theta_0$, so $\varphi \vdash r\theta_0$. We also have that v' is public w.r.t. φ'. We have $v'\sigma' = (u[y]_p)\sigma' = u\sigma'[y\sigma']_p = u\sigma[r\theta_0]_p$. Hence $u\sigma \to v'\sigma'$.
   From $t\sigma = l\theta_0$ and $t\sigma(^{M}/_{\mathtt{s}}) = l\theta$, we deduce that $\theta_0[^{M}/_{\mathtt{s}}] = \theta$, hence $r\theta_0[^{M}/_{\mathtt{s}}] = r\theta$. Thus $v'\sigma'(^{M}/_{\mathtt{s}}) = u\sigma(^{M}/_{\mathtt{s}})[r\theta]_p = v$.
   Since there is some $y \in \mathrm{dom}(\varphi)$ above z_0, we have that then $r\theta_0$ is a subterm of φ. Since φ is well-formed, we deduce that $r\theta_0$ satisfies the conditions of Definition 1. So φ' is also well-formed.

Claim 1: Let us now prove that there exists θ_0 such that $t\sigma = l\theta_0$. Otherwise we should have one of the following cases:

1. there is a position in l which is not a position in $t\sigma$;
2. there is a variable z in l having at least two occurrences, say at positions p_1, p_2, for which $t\sigma|_{p_1} \neq t\sigma|_{p_2}$.

Let us examine in detail the two cases:

1. Consider a minimal position (w.r.t. the prefix order) in l which is not a position in $t\sigma$. Then at the predecessor position an $\mathtt{s}$ occurs (since minimal

[3] For our equational theory there is exactly one occurrence of z_0 in l.

positions in l must be positions in $t\sigma(^{M}/_{\mathtt{s}})$, but not in $t\sigma$). This position is not ϵ (*i.e.* it does not correspond to the head of l) since otherwise M would not be in normal form. Now, for all other cases, by examining all rules in $\mathcal{R}_E$, we observe that at least one of Conditions 2 or 3 of Definition 1 (of well-formed frames) is not satisfied, which contradicts the hypothesis that φ is a well-formed frame.

2. Let $t_1 = t\sigma|_{p_1}$ and $t_2 = t\sigma|_{p_2}$. We have $t_1 \neq t_2$, but $t_1(^{M}/_{\mathtt{s}}) = t_2(^{M}/_{\mathtt{s}})$.
   We can have the following cases, according to whether the positions p_1 and p_2 are positions of t or not:
   (a) If p_1 and p_2 are positions of t. Then we can define $w_1 = t|_{p_1}$ and $w_2 = t|_{p_2}$. We have $w_1\sigma \neq w_2\sigma$, but $w_1\sigma(^{M}/_{\mathtt{s}}) = w_2\sigma(^{M}/_{\mathtt{s}})$. Since w_1 and w_2 are public, the disequality is contradicted by Lemma 1.
   (b) If p_1 is not a position of t. Let p_y be the position in t such that $p_y < p_1$ and $t|_{p_y} = y$ for some $y \in \mathrm{dom}(\sigma)$.
   – A special case is when the rule $\mathrm{check}(z_1, \mathrm{sign}(z_1, \mathrm{priv}(z_2)), \mathrm{pub}(z_2)) = \mathrm{ok}$ is applied with $z = z_1$.
   Since the positions of z_1 in l are 1 and $2 \cdot 1$, and $p_y < p_1$ we have that $p_1 = 2 \cdot 1$, $p_2 = 1$ and $p_y = 2$ ($p_y = \epsilon$ implies that σ contains a destructor symbol). Hence $t\sigma|_{p_1} = y\sigma|_1$. Using the equality $\mathrm{retrieve}(\mathrm{sign}(z_1, z_2)) = z_1$ we notice that $y\sigma|_1$ is actually equal to $\mathrm{retrieve}(y\sigma)$. Considering $w_1 = t|_1$ and $w_2 = \mathrm{retrieve}(y)$, we have $w_1\sigma(^{M}/_{\mathtt{s}}) = w_2\sigma(^{M}/_{\mathtt{s}})$. Since w_1 and w_2 are public, this implies by Lemma 1 that $w_1\sigma = w_2\sigma$ thus $t_1 = t_2$, a contradiction.
   – Otherwise, by examining all the other cases and using the fact that φ is well-formed, we verify that $t' = t\sigma|_{p_1}$ is a subterm of σ that does not contains $\mathtt{s}$. Now either p_2 is also not a position of t, then symmetrically $t|_{p_2}$ does not contain $\mathtt{s}$ hence $t_1 = t_1(^{M}/_{\mathtt{s}}) = t_2(^{M}/_{\mathtt{s}}) = t_2$, a contradiction. Or p_2 is a position of t, then $t|_{p_2}$ is a public term, and the disequality is contradicted by (the second part of) Lemma 1.

B Active Case

B.1 Proof of Lemma 3

Lemma 3. *Let P be a closed plain process, and A be a closed extended process such that $P \Rightarrow^* A$. There are $k \geq 0$, an extended process $B = \nu\widetilde{n}.\sigma_k|P_B$, where P_B is some plain process, and θ a substitution public w.r.t. $\widetilde{n}$ such that: $A \equiv B$, $\{\widetilde{n}\} \subseteq \mathrm{bn}(P)$, for every side of a test or an output M of P_B there is a message M_0 in P (a side of a test or an output respectively, such that $M = M_0\theta\sigma_k$, and, $\sigma_i = \sigma_{i-1} \cup \{^{m_i\theta_i\sigma_{i-1}}/_{y_i}\}$, for all $i \in \{1, \ldots, k\}$, where m_i is an output in P, θ_i is a substitution public w.r.t. $\widetilde{n}$ and σ_0 is the empty substitution.*

Proof. We provide an inductive and constructive proof. We reason by induction on the number of reductions in $P \Rightarrow^* A$.

The base case is evident.

Assume that $P \Rightarrow^l A_l$ and that there are k, B_l and θ as in the statement of the lemma. Suppose that $A_l \Rightarrow A_{l+1}$ and regard what kind of reduction rule was used in this last step:

- If it is an internal reduction then, since static equivalence is closed by structural equivalence and by internal reduction (see Lemma 1 in [1]), it is sufficient to consider as searched values the same as for A_l.
- If it is a labeled reduction then we prove the following property: $\alpha \neq \overline{c}\langle x\rangle$ (for any a and x) and there is an extended process $B_{l+1} = \varphi(B_{l+1})|P_{l+1}$ such that $B_{l+1} \equiv A_{l+1}$ and
  - if $\alpha = \nu x.\overline{c}\langle x\rangle$ then $P_{l+1} = P_l$ and $\varphi(B_{l+1}) = \nu\widetilde{n}.\sigma_{k+1}$, where $\sigma_{k+1} = \sigma_k \cup \{^{M_l}/_x\}$ and M_l is an output in P_l.
  - if $\alpha = c(M)$ then $\varphi(B_{l+1}) = \varphi(B_l)$ and for every message (a side of a test or an output) M_{l+1} in P_{l+1} there is a message (a side of a test or an output, respectively) M_l in P_l, such that $M_{l+1} = M_l\theta'\sigma_k$, for some substitution θ' public w.r.t. $\nu\widetilde{n}$.
  - if $\alpha = \overline{c}\langle n\rangle$ or $\alpha = \nu n.\overline{c}\langle n\rangle$ then $P_{l+1} = P_l$, and $\varphi(B_{l+1}) = \varphi(B_l)$ or $\varphi(B_{l+1}) = \nu\{\widetilde{n}\}\backslash\{n\}.\sigma_k$, respectively.

  It is easy to see that this property is sufficient to prove the inductive step. The property can be verified, by showing, using induction on the shape of the derivation tree, that for any extended processes A', A'', B' such that $A' \xrightarrow{\alpha} A''$, $A' \equiv B'$, $B' = \nu\widetilde{n}.\sigma|Q$ there is B'' such that $A'' \equiv B''$ and $B' = \nu\widetilde{n}'.\sigma'|Q'$ where
  - if $\alpha = c(M)$ then $\widetilde{n}' = \widetilde{n}$, $\sigma' = \sigma$ and $N'' = N'\{^{M}/_x\}$ for each term N'' of B'' where N' is the corresponding term in B' and $c(x)$ is an input in B';
  - if $\alpha = \nu x.\overline{c}\langle x\rangle$ then $Q' = Q$, $\widetilde{n}' = \widetilde{n}$, and $\sigma' = \sigma \cup \{^{M}/_x\}$ where $\overline{c}\langle M\rangle$ is an input in B';
  - if $\alpha = \overline{c}\langle x\rangle$, $\alpha = \overline{c}\langle n\rangle$ or $\alpha = \nu n.\overline{c}\langle n\rangle$ then $\widetilde{n}' = \widetilde{n}$ for the first two cases, and $\{\widetilde{n}'\} = \{\widetilde{n}\}\backslash\{n\}$ for the third one, $\sigma' = \sigma$ and $Q' = Q$.

B.2 Passive case revisited

We have to generalize our result to extended well-formed frames.

Proposition 1. *Let $\varphi = \nu(\widetilde{n} \uplus \{\mathtt{s}\}).\sigma$ be an extended well-formed frame w.r.t $\mathtt{s}$. $\varphi \not\vdash \mathtt{s}$ iff $\nu\widetilde{n}.\sigma(^{M}/_{\mathtt{s}}) \approx \nu\widetilde{n}.\sigma(^{M'}/_{\mathtt{s}})$ for all M, M' closed public terms w.r.t. φ.*

As for the proof of Theorem 1, we first proof some lemmas similar to Lemmas 1 and 2.

Lemma 7. *Let $\varphi = \nu\widetilde{n}.\sigma$ be an extended well-formed frame w.r.t. $\mathtt{s} \in \widetilde{n}$. If $\varphi \not\vdash \mathtt{s}$ then for all public terms u, v, M w.r.t. $\widetilde{n}$, M being ground, $u\sigma(^{M}/_{\mathtt{s}}) = v\sigma(^{M}/_{\mathtt{s}})$ implies $u\sigma = v\sigma$.*

Proof. Suppose that $u\sigma(^{M}/_{\mathtt{s}}) = v\sigma(^{M}/_{\mathtt{s}})$ and $u\sigma \neq v\sigma$. Then there is an occurrence p of $\mathtt{s}$, suppose in $u\sigma$, such that $v\sigma|_p \neq \mathtt{s}$. Consider the variable $y \in \mathcal{V}(u) \subseteq \mathrm{dom}(\sigma)$ and its occurrence p_y in u such that $p_y \leq p$. Let $p' = \mathrm{sf}(p, p_y)$.

Since φ is an extended well-formed frame, we have that there is an agent encryption at occurrence q in $y\sigma$ such that $q \leq p'$. Hence the term at position $q.3$ is a restricted name. It results that $q.3 \notin \mathrm{Pos_v}(v)$, since v is public. That is there is variable $y' \in \mathcal{V}(v) \subseteq \mathrm{dom}(\sigma)$ with the occurrence $p_{y'}$ such that $p_{y'} \leq p_y.q.3$. Let $m = y\sigma$ and $m' = y'\sigma$. Let $q' = \mathrm{sf}(p_y.q, p_{y'})$. Since $m|_{q.3} = m'|_{q'.3}$, we have, by unicity of the randomness in agent encryptions, that $m|_q = m'|_{q'}$. This means in particular that $m|_{p'} = m'|_{p''}$, where $p'' = q'.\mathrm{sf}(p', q)$. But since $m|_{p'} = \mathtt{s}$ and $p = p_{y'}.q'.\mathrm{sf}(p', q)$, this contradicts the fact that $v\sigma|_p \neq \mathtt{s}$.

The following lemma is proved similarly.

Lemma 8. *Let $\varphi = \nu\widetilde{n}.\sigma$ be an extended well-formed frame w.r.t. $\mathtt{s} \in \widetilde{n}$ such that $\varphi \not\vdash \mathtt{s}$, u be a subterm of a term of σ such that $\sigma \cup \{^{u}/_{y}\}$ is still a well-formed frame, and v be a public term w.r.t. $\widetilde{n}$. Then, for all public ground term M, $u(^{M}/_{\mathtt{s}}) = v\sigma(^{M}/_{\mathtt{s}})$ implies $u = v$.*

The following lemma is similar to Lemma 2.

Lemma 9. *Let $\varphi = \nu\widetilde{n}.\sigma$ be an extended well-formed frame w.r.t. $\mathtt{s} \in \widetilde{n}$ such that $\varphi \not\vdash \mathtt{s}$ and u, M public terms w.r.t. $\widetilde{n}$, M being ground and in normal form. If $u\sigma(^{M}/_{\mathtt{s}}) \to v$, then there exists an extended well-formed frame $\varphi' = \nu\widetilde{n}.\sigma'$ such that*

- $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma')$, $y\sigma' = y\sigma, \forall y \in \mathrm{dom}(\sigma)$,
- *for all term w, $\varphi \vdash w$ iff $\varphi' \vdash w$,*
- *and there exists a public term v' w.r.t. $\widetilde{n}$ such that $v = v'\sigma'$.*

Proof. Let u, v, M be terms such that $u\sigma(^{M}/_{\mathtt{s}}) \to v$, as in the statement of the lemma. Let $l \to r \in \mathcal{R}_E$ be the rule that was applied in the above reduction and p be the position at which it was applied, i.e. $u\sigma(^{M}/_{\mathtt{s}})|_p = l\theta$. Since M is in normal form, p must be a position of $u\sigma$.

Assume that there is a substitution θ_0 such that $u\sigma|_p = l\theta_0$. This will be proved later. Since $\varphi = \nu\widetilde{n}.\sigma$ be an extended well-formed frame, we know there is an agent encryption above $\mathtt{s}$ at position $q_{\mathrm{enc}} < p$ such that there is only pairing along the path between q_{enc} and p. We deduce that $p \in \mathrm{Pos_{nv}}(u)$. So let $t = u|_p$. We have $t\sigma(^{M}/_{\mathtt{s}}) = l\theta$.

For our equational theory, r is a variable z_0.

Consider the[4] position q of z_0 in l. The position q is also in $l\theta_0$, that is, in $t\sigma$.

1. If q is a position in t but not in $t\sigma$ (that is, there is no y_i above z_0) then take $v' = u[t|_q]_p$ and $\sigma' = \sigma$. It is easy to verify that the conditions of the Lemma 9 are satisfied.
2. If q is a position in $t\sigma$, but not in t (that is, there is a y_i above z_0). Then take $v' = u[y]_p$ and $\sigma' = \sigma \cup \{r\theta_0/y\}$, where $y \notin \mathrm{dom}(\sigma)$. We have that $t\sigma =_E r\theta_0$, so $\varphi \vdash r\theta_0$. We also have that v' is public w.r.t. φ'.
   We have $v'\sigma' = (u[y]_p)\sigma' = u\sigma'[y\sigma']|_p = u\sigma[r\theta_0]_p$. And hence $u\sigma \to v'\sigma'$.

[4] For our equational theory there is exactly one occurrence of z_0 in l.

¿From $t\sigma = l\theta_0$ and $t\sigma(M/\mathtt{s}) = l\theta$, we deduce that $\theta_0(M/\mathtt{s}) = \theta$ hence $r\theta_0(M/\mathtt{s}) = r\theta$. Thus $v'\sigma'(M/\mathtt{s}) = u\sigma(M/\mathtt{s})[r\theta]_p = v$.
Since there is a y_i above z_0, we have that then $r\theta_0$ is a subterm of φ. Since φ is an extended well-formed frame and $\varphi \nvdash \mathtt{s}$, we deduce that $r\theta_0$ verifies the condition of well-formedness. Thus φ' is an extended well-formed frame.

Let us now prove that there exists indeed a θ_0 such that $t\sigma = l\theta_0$. Assume by contradiction that it is not the case. At least one of the following cases must occur:

1. there is a position in l which is not a position in $t\sigma$;
2. there is a variable z in l having at least two occurrences, say at positions p_1, p_2 in l, for which $t\sigma|_{p_1} \neq t\sigma|_{p_2}$.

Let us examine in detail the two cases:

1. This is in fact an impossible case. Indeed, φ is an extended well-formed frame and $\varphi \nvdash \mathtt{s}$, it must be the case that $l = \mathrm{dec}(\mathrm{enc}(z_0, z_2, z_3), z_2)$ but since there is at least one encryption above $\mathtt{s}$, all positions of l are in $t\sigma$.
2. Again, it must be the case that $l = \mathrm{dec}(\mathrm{enc}(z_0, z_2, z_3), z_2)$.
   (a) Either both p_1 and p_2 are both positions in t. Then we can consider $w_1 = t|_{p_1}$ and $w_2 = t|_{p_2}$. We have $w_1\sigma \neq w_2\sigma$, but $w_1\sigma(M/\mathtt{s}) = w_2\sigma(M/\mathtt{s})$. Since w_1 and w_2 are public, the inequality is contradicted by Lemma 7.
   (b) Or $p_1, p_2 \notin \mathrm{Pos}(t)$. Let p_y be the position in t such that $p_y < p_1$ and $t|_{p_y} = y$ for some $y \in \mathrm{dom}(\sigma(\mathtt{s}))$. applied We must have that $p_1 = 1.2$, $p_2 = 2$ and $p_y = 1$. Hence $t\sigma|_{p_1} = y\sigma|_2$, that is, it is a subterm $t|_2$ of a term of σ, and $t\sigma|_{p_2} = t|_2)\sigma$. $t|_2$ being a public term, we can apply Lemma 8 and derive a contradiction.

The proof of Proposition 1 ends like the proof of Theorem 1.

B.3 Proof of the main result

Let u, v be two terms. Define $\mathrm{Pos}(u, v) = \{p \in \mathrm{Pos}(u) \mid u|_p = v\}$.

We denote by $u \to^q v$ the reduction $u \to v$ such that $u|_q = l\theta$ and $v = u[r\theta]_q$, where q is a position in u, a rule $l \to r \in \mathcal{R}_E$, and θ is a substitution. Consider a position p in u. The function nfp_1 computes the corresponding position in v of the function symbol (or variable or name) at position p in u. Accordingly, the function nfp computes the corresponding position in $u{\downarrow}$. The function nfp^{-1} will do the opposite: to a position in $u{\downarrow}$ it associates the corresponding position in u. We say that a function symbol at position p is *consumed w.r.t. the reduction* $u \to^q v$ if $\mathrm{nfp}_1(u, p, q)$ is undefined. Similarly, we say that the same occurrence is *consumed w.r.t. the normal form* $u{\downarrow}$ if $\mathrm{nfp}(u, p)$ is undefined. We will say only that an occurrence is consumed when it is clear from the context which definition is used. Formally, we define the function $\mathrm{nfp}_1 : \mathcal{T} \times \mathbb{N}_+^* \times \mathbb{N}_+^* \hookrightarrow \mathbb{N}_+^*$

$$\mathrm{nfp}_1(u, p, q) = \begin{cases} p', & \text{if } u \to^q v \\ \bot, & \text{otherwise,} \end{cases}$$

where

$$p' = \begin{cases} p, & \text{if } p \not\geq q, \\ \bot, & \text{if } p \geq q \ \wedge \ p \not\geq q.q_r, \\ q.\,\mathrm{sf}(p, q.q_r), & \text{if } p \geq q.q_r, \end{cases}$$

where $l \to r$ is the rule that was applied and q_r is the position of r in l. Observe that for the equational theory E there's at most one rule that can be applied and there's exactly one occurrence of r in l. The function $\mathrm{nfp}\colon \mathcal{T} \times \mathbb{N}_+^* \hookrightarrow \mathbb{N}_+^*$ is defined by $\mathrm{nfp}(u,p) = p_k$ where $u \to^{q_1} \cdots \to^{q_k} u_k$, $u_k = u{\downarrow}$, $p_i = \mathrm{nfp}_1(u, p_{i-1}, q_i)$, for $1 \leq i \leq k$ and $p_0 = p$. The definition is correct since $\mathcal{R}_E$ is convergent. We define $\mathrm{nfp}^{-1}\colon \mathcal{T} \times \mathbb{N}_+^* \hookrightarrow \mathbb{N}_+^*$, $\mathrm{nfp}^{-1}(u,p) = p'$ iff $\mathrm{nfp}(u,p') = p$.

Lemma 4. *Let P be a well-formed process with no test over $\mathtt{s}$ and $\varphi = \nu\widetilde{n}.\sigma$ be a valid frame w.r.t. P such that $\nu\mathtt{s}\varphi \nvdash \mathtt{s}$. Consider the corresponding standard frame $\nu\widetilde{n}.\overline{\sigma} = \nu\widetilde{n}.\{t_j \mid 1 \leq j \leq k\}$. For every occurrence $q_\mathtt{s}$ of $\mathtt{s}$ in $t_j{\downarrow}$, we have $f_e(t_j{\downarrow}, q_\mathtt{s}) = e[{}^{w}/_{\mathtt{x}}]$ for some $e \in \mathcal{E}$ and some term w. In addition $\nu\widetilde{n}.\sigma_j{\downarrow}$ is an extended well-formed frame w.r.t. $\mathtt{s}$.*

Proof. We reason by induction on j.

Base case: $j = 1$. We have that $t_1 = m_1\theta_1$. The position $q_\mathtt{s}$ in fact a position in m_1 $\mathtt{s}$ can't appear in θ since $\mathtt{s}$ is restricted and θ is a public substitution. There must an encryption above $\mathtt{s}$ in m_1, since otherwise $\mathtt{s}$ would be deducible. Then the result follows immed iately from the properties of well-formed processes and the definition of $\mathcal{E}_0$ (take $w = \mathtt{s}$).

Inductive step. Let $p_\mathtt{s} = \mathrm{nfp}^{-1}(t_j, q_\mathtt{s})$. If $p_\mathtt{s}$ is in m_j then, as in the previous paragraph, $f_e(t_j{\downarrow}, q_\mathtt{s})[{}^{\mathtt{x}}/_{\mathtt{s}}] \in \mathcal{E}_0$.

Let $p_\mathtt{s} = \mathrm{nfp}^{-1}(t_j, q_\mathtt{s})$. If $p_\mathtt{s}$ is in m_j then, as in the previous paragraph, $f_e(t_j{\downarrow}, q_\mathtt{s})[{}^{\mathtt{x}}/_{\mathtt{s}}] \in \mathcal{E}_0$.

If $p_\mathtt{s}$ is in σ_{j-1}, then let z be the variable in m_j at position say p_z, where $p_z < p_\mathtt{s}$ and let y_{j_1} be the variable of $z\theta_j$ on the path to $p_\mathtt{s}$ at position say p_{y_1}. We have that $j_1 \leq j - 1$. Let $p^1_\mathtt{s} = \mathrm{sf}(p_\mathtt{s}, p_{y_1})$ and $q^1_\mathtt{s} = \mathrm{nfp}(t_{j_1}, p^1_\mathtt{s})$. By recursion hypothesis, σ_{j-1} is a well-formed frame and $f_e(t_{j_1}{\downarrow}, q^1_\mathtt{s}) = e[{}^{w}/_{\mathtt{x}}]$ with $e \in \mathcal{E}_l$, for some term w and some $l \geq 0$. It follows that $q^1_{\mathrm{enc}} = \max\{\, q \in \mathrm{Pos}(t_{j_1}{\downarrow}) \mid q < q_\mathtt{s} \ \wedge \ h_{(t_{j_1}{\downarrow})|_q} = \mathrm{enc} \,\}$ exists. Let $p^1_{\mathrm{enc}} = \mathrm{nfp}^{-1}(t_{j_1}, q^1_{\mathrm{enc}})$.

If $p_{y_1}.p^1_{\mathrm{enc}}$ is not consumed in $t_j{\downarrow}$ then it follows that $\mathrm{nfp}(t_j, p_{y_1}.p^1_{\mathrm{enc}})$ is the lowest encryption in $t_j{\downarrow}$ (since it corresponds to q^1_{enc}). It follows that $f_e(t_j{\downarrow}, q_\mathtt{s}) = f_e(t_{j_1}{\downarrow}, q^1_\mathtt{s})$.

If $p_{y_1}.p^1_{\mathrm{enc}}$ is consumed in $t_j{\downarrow}$, consider the occurrence of dec in t_j, say p_{dec}, that consumes it. Since p^1_{enc} is not consumed in $t_{j_1}{\downarrow}$ it follows that p_{dec} is in $z\theta_j$ or in m_j, and all encryptions above p^1_{enc} in t_{j_1} are consumed in t_j. If p_{dec} is in $z\theta_j$ then all encryptions above p^1_{enc} in t_{j_1} are consumed by decryptions that are in $z\theta_j$. This means that in $(z\theta_j\sigma_{j-1}){\downarrow}$ there's no encryption above $\mathtt{s}$, and in particular no agent encryption, which contradicts that σ_{j-1} is a encryption above extended well-formed frame. Hence p_{dec} is in m_j.

Let u, v, k, k', n be terms such that $\mathrm{dec}(u,k) = t_j|_{p_{\mathrm{dec}}}$ and $\mathrm{enc}(v,k',n) = t_j|_{p_{y_1}.p^1_{\mathrm{enc}}}$. We have that $k =_E k'$ since p_{dec} consumes $p_{y_1}.p^1_{\mathrm{enc}}$. Since p_{dec} is from

the output m_j and p^1_{enc} is also from an output being an agent encryption we have that k and k' are in normal form, hence $k = k'$. We then have $\text{dec}(u, k) \rightarrow^* \text{dec}(\text{enc}(v, k, n), k) \rightarrow^* v\downarrow$.

Let $(d, p) = f_d(m, p_z)$ and consider d_i such that the decryption p_{dec} is in d_i. Since $\mathtt{s}$ is in $t_j\downarrow$ it follows that $\mathtt{x}$ is in $d_i(e)\downarrow$. From the first condition of processes that do not test over $\mathtt{s}$ we have that $i = 1$ and $\overline{e} \not<_{st} d_1$. Since p_{dec} consumes $p_{y_1}.p^1_{\text{enc}}$, above p_{dec} in d_1 there are only projections, below enc in e there are only pairs and $\overline{e} \not<_{st} d_1$ it follows that $d_1 \leq_{st} \overline{e}$. Hence $d_1 \in \overline{\mathcal{E}}_l$.

Suppose that there is no encryption above p_{dec} in m_j. Then since d_1 is consumed and above d_1 in m_j there are only pairs, it follows that $\mathtt{s}$ is deducible from σ_j (t_j that is). Thus there is at least one encryption above p_{dec} in m_j. Let p_{enc} be the lowest decryption above p_{dec} in m_j. And let $(m', p'_{\text{enc}}) = f_{ep}(m_j, p_z)$. Then $m'[\mathtt{x}]_p \in \mathcal{E}_{l+1}$.

Since p_{enc} is not consumed in $t_j\downarrow$ and in m' all function symbols above p are not destructors we have that $f_e(t_j, p_{\mathtt{s}}) \rightarrow^* (m'[\mathtt{x}]_p)[\mathtt{x} \rightarrow d_1(f_e(\text{enc}(v, k, n), p'_{\mathtt{s}}))]$ where $p'_{\mathtt{s}} = \text{sf}(p^1_{\mathtt{s}}, p^1_{\text{enc}})$. Hence $f_e(t_j\downarrow, q_{\mathtt{s}}) = (m'[\mathtt{x}]_p)[^{w'}/_{\mathtt{x}}]$, where $w' = d_1(f_e(\text{enc}(v, k, n), p'_{\mathtt{s}})\downarrow$. That is we have the first part of the lemma.

In order to prove that $\sigma\downarrow$ is a well-formed frame we show that $m'[\mathtt{x}]_p$ and w' contain only pairs as function symbols, except for the head of $m'[\mathtt{x}]_p$ which is an encryption. We have that all function symbols, except the head in $m'[\mathtt{x}]_p$, are pairs (it follows from the definition of m'). The term w' is a subterm of $f_e(\text{enc}(v\downarrow, k, n), q'_{\mathtt{s}})$ which contains only pairs as function symbols (except for the head), since σ_{j_1} is well-formed frame.

Lemma 10. *Let P be a well-formed process with no test over $\mathtt{s}$, let $\varphi = \nu\widetilde{n}.\sigma$ a valid frame w.r.t. P such that $\varphi \nvdash \mathtt{s}$, and $T \in \mathcal{M}_t(P)$ a side of a test. Let θ a public substitution. If $T \notin \mathcal{M}^{\mathtt{s}}_t$ and $\mathtt{s} \in \text{fn}((T\theta\sigma)\downarrow)$ then $(T\theta\sigma)\downarrow = u\sigma'$ where σ' is an extended well-formed frame as in Lemma 9 and u is some term (not necessarily public).*

Proof. Suppose that $T \notin \mathcal{M}^{\mathtt{s}}_t$ and $\mathtt{s} \in \text{fn}(T\theta\sigma(\mathtt{s})\downarrow)$. Hence T is not ground and denote by z the variable of T and by p_z its position. Consider an occurrence $q_{\mathtt{s}}$ of $\mathtt{s}$ in $T\theta\sigma(\mathtt{s})\downarrow$. Denote $t_z = z\theta\sigma(\mathtt{s})\downarrow$. We then have that $\mathtt{s} \in \text{fn}(t_z)$.

Let $p_{\mathtt{s}} = \text{nfp}^{-1}(T\theta\overline{\sigma}(\mathtt{s}), q_{\mathtt{s}})$. Let y_j be the variable of $z\theta$ on the path to $p_{\mathtt{s}}$ at position say p_y, with $1 \leq j \leq k$ (see Lemma 3). Applying Lemma 4 to t_j we obtain that $f_e(t_j\downarrow, q_{\mathtt{s}}) = e[^w/_{\mathtt{x}}]$ with $e \in \mathcal{E}_l$, for some term w and some $l \geq 0$. Consider the lowest encryption q_{enc} in $t_j\downarrow$ above $q'_{\mathtt{s}}$, where $q'_{\mathtt{s}}$ is the corresponding positions of $q'_{\mathtt{s}}$ in $t_j\downarrow$. If this encryption is consumed then it must be consumed be a dec from T since otherwise $\mathtt{s}$ would be deducible. It follows that there is $1 \leq i \leq k$ such that $d_i = \pi^i(\text{dec}(\text{z}, k))$, where $f_d(T, p_z) = d_1(\ldots d_k)$ and $e = \text{enc}(u, k, r)$. Moreover $\mathtt{x} \in d_i(e)\downarrow$. Thus $T \in \mathcal{M}^{\mathtt{s}}_t$, but this contradicts the supposition. Hence q_{enc} is not consumed in $T\theta\sigma(\mathtt{s})\downarrow$. Then it is sufficient to consider the position $\text{nfp}^{-1}(t_j, q_{\text{enc}})$ (it is in some σ_{j_1}) in t_j in order to find the required u and σ'.

Lemma 5. *Let P be a well-formed process with no test over $\mathtt{s}$, $\varphi = \nu\widetilde{n}.\sigma$ a valid frame for P such that $\nu\mathtt{s}\varphi \not\vdash \mathtt{s}$ and θ a public substitution. If $T_1 = T_2$ is a test in P, then $T_1\theta\sigma(^{M}/_{\mathtt{s}}) =_E T_2\theta\sigma(^{M}/_{\mathtt{s}})$ implies $T_1\theta\sigma =_E T_2\theta\sigma$.*

Proof. We say a test T is in case A, B or C if

- there is no $\mathtt{s}$ in $T\theta\sigma(\mathtt{s})\downarrow$,
- there is $\mathtt{s}$ in $T\theta\sigma(\mathtt{s})\downarrow$, $T \notin \mathcal{M}_t^{\mathtt{s}}$, or
- there is $\mathtt{s}$ in $T\theta\sigma(\mathtt{s})\downarrow$, $T \in \mathcal{M}_t^{\mathtt{s}}$, respectively.

Suppose that $T_1\theta\sigma(^{M}/_{\mathtt{s}})\downarrow = T_2\theta\sigma(^{M}/_{\mathtt{s}})\downarrow$ and $T_1\theta\sigma(\mathtt{s})\downarrow \neq T_2\theta\sigma(\mathtt{s})\downarrow$. We consider all possible cases T_1 and T_2 could be in:

- AA. The supposition is clearly false.
- BA, BB. By Lemma 10 we have that $(T_1\theta\sigma(\mathtt{s}))\downarrow = u\sigma'(\mathtt{s})$. Suppose there is an occurrence of $\mathtt{s}$ in $(T_1\theta\sigma(\mathtt{s}))\downarrow$ such that the term at the corresponding position in $(T_2\theta\sigma(\mathtt{s}))\downarrow$ is not $\mathtt{s}$. There is an agent encryption $\mathrm{enc}(v, w, n)$ above $\mathtt{s}$ in $(T_1\theta\sigma(\mathtt{s}))\downarrow$. The name n in $(T_2\theta\sigma(\mathtt{s}))\downarrow$ may come from $\sigma(\mathtt{s})$, from θ or from T_2. But it cannot come from T_2 (see the definition of well-formed processes), it cannot come from θ since n is restricted and θ is public, and it cannot come from σ since σ is well-formed (and hence encryption is probabilistic).
- CA, CB, CC. Since $T_1 \in \mathcal{M}_t^{\mathtt{s}}$, condition 2 of processes that do not test over $\mathtt{s}$ says that T_2 is a restricted name. Thus T_2 cannot be in cases B or C: since $\mathtt{s}$ doesn't appear in tests, T_2 should be non ground. If T_2 is in case A then there is a contradiction since T_2 should be a subterm of M but this is impossible since M is public, while T_2 is restricted.

Theorem 2. *Let P be well-formed process w.r.t. a free name $\mathtt{s}$, which is not a channel name, such that P does not test over $\mathtt{s}$. We have $\nu\mathtt{s}\varphi \not\vdash \mathtt{s}$ for any valid frame φ w.r.t. P if and only if $P(^{M}/_{\mathtt{s}}) \approx_l P(^{M'}/_{\mathtt{s}})$, for all ground terms M, M' public w.r.t.* $\mathrm{bn}(P)$.

Proof. Consider the relation $\mathcal{R}$ between extended processes defined as follows: $A\,\mathcal{R}\,B$ if there is an extended process $A_0(\mathtt{s})$ such that $P(\mathtt{s}) \Rightarrow^* A_0(\mathtt{s})$ and ground terms M, M' public w.r.t. $\nu(\widetilde{n} \cup \{\mathtt{s}\})$ such that $A = A_0(^{M}/_{\mathtt{s}})$ and $B = A_0(^{M'}/_{\mathtt{s}})$.

We show that $\mathcal{R}$ satisfies the three points of the definition of labeled bisimilarity. Suppose $A\,\mathcal{R}\,B$, that is $A_0(^{M}/_{\mathtt{s}})\,\mathcal{R}\,A_0(^{M'}/_{\mathtt{s}})$ for some A_0, M, M' as above. In what follows we write $X(t)$ for $X(^{t}/_{\mathtt{s}})$, where X ranges over processes and frames and t is M or M'. We prove that the following questions have affirmative answer:

1. $\varphi(A_0(M)) \approx \varphi(A_0(M'))$? We know that $\varphi(A_0(\mathtt{s}))$ is a valid frame (from the definition of $\mathcal{R}$), hence $\varphi(A_0(\mathtt{s})) \not\vdash \mathtt{s}$ (from the hypothesis). Let $\varphi'(\mathtt{s}) \equiv \varphi(A_0(\mathtt{s}))$ having only ground and normalized terms. Then, by Lemma 4, we have that $\varphi'(\mathtt{s})$ is an extended well-formed frame. We can then use Proposition 1 to obtain that $\varphi(A_0(M)) \approx \varphi(A_0(M'))$, since we have $\varphi(A_0(M)) = \varphi(A_0(\mathtt{s}))(M)$ (and the same for M').

2. if $A_0(M) \to A'$ then $A' \equiv A'_0(M)$, $A_0(M') \to A'_0(M')$ and $A'_0(M)\,\mathcal{R}\,A'_0(M')$, for some A'_0? We distinguish two cases, according to whether the used rule was the COMM rule or one of the THEN and ELSE rules:
   - if the COMM rule was used then $A_0(M) \equiv C(M)[\overline{c}\langle z\rangle.Q(M)|c(z).R(M)]$, where C is an evaluation context and $A' = C(M)[Q(M)|R(M)]$. Then $A_0(\mathtt{s}) \equiv C(\mathtt{s})[\overline{c}\langle z\rangle.$
   $Q(\mathtt{s})|c(z).R(\mathtt{s})]$. Take $A'_0(\mathtt{s}) = C(\mathtt{s})[Q(\mathtt{s})|R(\mathtt{s})]$. We have that $P(\mathtt{s}) \Rightarrow^* A'_0(\mathtt{s})$ and so, by definition of $\mathcal{R}$, we have that $A'_0(M)\,\mathcal{R}\,A'_0(M')$.
   - otherwise, $A_0(M) \equiv C(M)[\text{if } T'(M) = T''(M) \text{ then } Q(M) \text{ else } R(M)]$. Then $A_0(\mathtt{s}) \equiv C(\mathtt{s})[\text{if } T'(\mathtt{s}) = T''(\mathtt{s}) \text{ then } Q(\mathtt{s}) \text{ else } R(\mathtt{s})]$. From Lemma 3 we know that $T'(\mathtt{s}) = T'_0\theta\sigma(\mathtt{s})$ and $T''(\mathtt{s}) = T''_0\theta\sigma(\mathtt{s})$, where $T'_0 = T''_0$ is a test in P and $\nu\widetilde{n}.\sigma \equiv \varphi(A_0(\mathtt{s}))$ is the standard frame w.r.t. $A_0(\mathtt{s})$. Take $A'_0(\mathtt{s}) = C(\mathtt{s})[Q(\mathtt{s})]$ if $T'_0\theta\sigma(\mathtt{s}) =_E T''_0\theta\sigma(\mathtt{s})$ and $A'_0(\mathtt{s}) = C(\mathtt{s})[R(\mathtt{s})]$ otherwise. From Lemma 5 we have that $T'_0\theta\sigma(\mathtt{s}) =_E T''_0\theta\sigma(\mathtt{s})$ iff $T'_0\theta\sigma(M) =_E T''_0\theta\sigma(M)$. Hence $A_0(M) \to A'_0(M)$, $A_0(M') \to A'_0(M')$ and $A_0(\mathtt{s}) \to A'_0(\mathtt{s})$. And we also have $A'_0(M)\,\mathcal{R}\,A'_0(M')$ from the definition of $\mathcal{R}$.
3. if $A_0(M) \xrightarrow{\alpha} A'$ and $\mathrm{fv}(\alpha) \subseteq \mathrm{dom}(\varphi(A_0(M)))$ and $\mathrm{bn}(\alpha) \cap \mathrm{fn}(A_0(M')) = \emptyset$ then $A' \equiv A'_0(M)$, $A_0(M') \xrightarrow{\alpha} A'_0(M')$ and $A'_0(M)\,\mathcal{R}\,A'_0(M')$, for some A'_0? According to the form of α, we consider the following cases:
   - $\alpha = c(T)$. Suppose $A_0(M) \equiv C(M)[c(z).Q(M)]$. Then take $A'_0(\mathtt{s}) = C(\mathtt{s})[Q(\mathtt{s})$
   $\{^T/_z\}]$.
   - $\alpha = \overline{c}\langle u\rangle$. Suppose $A_0(M) \equiv C(M)[\overline{c}\langle u\rangle.Q(M)]$. Then take $A'_0(\mathtt{s}) = C(\mathtt{s})[Q(\mathtt{s})]$.
   - $\alpha = \nu u.\overline{c}\langle u\rangle$. Suppose $A_0(M) \equiv C(M)[\nu u.A_1(M)]$, where $A_1(M) \xrightarrow{\overline{c}\langle u\rangle} A'_1(M)$. Then take $A'_0(\mathtt{s}) = C(\mathtt{s})[A_1(\mathtt{s})]$.

The above discussion proves that $\mathcal{R} \subseteq \approx_l$. Since we have clearly that $P(^M/_{\mathtt{s}})\,\mathcal{R}\,P(^{M'}/_{\mathtt{s}})$, it follows that $P(^M/_{\mathtt{s}}) \approx_l P(^{M'}/_{\mathtt{s}})$.

C Examples

For sake of simplicity, we may omit the symbol $\langle,\rangle$ for pairing. In that case, we assume a right priority that is $a, b, c = \langle\langle a, b\rangle, c\rangle$.

C.1 Needham-Schroeder symmetric key protocol

The protocol is described below:

$$\begin{array}{l} A \Rightarrow S : A, B, N_a \\ S \Rightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}} \\ A \Rightarrow B : \{K_{ab}, A\}_{K_{bs}} \end{array}$$

Our target secret is K_{ab}.

The corresponding process is:

$$P_{NS}(k_{ab}) = \nu k_{as}.\nu k_{bs}.(!A)|(!c(z_b))|(!\nu k.S(k))|S(k_{ab})$$

where

$$\begin{aligned} A = \nu n_a.\overline{c}\langle a, b, n_a\rangle.c(z_a).[\pi_1(\mathrm{dec}(z_a, k_{as})) = n_a]. \\ [\pi_1(\pi_2(\mathrm{dec}(z_a, k_{as}))) = b].\overline{c}\langle\pi_2(\pi_2(\pi_2(\mathrm{dec}(z_a, k_{as}))))\rangle \\ S(x) = c(z_s).\nu r, r'.\overline{c}\langle\mathrm{enc}(\langle\pi_2(\pi_2(z_s)), \pi_1(\pi_2(z_s)), k_{ab}, \\ \mathrm{enc}(\langle x, \pi_1(z_s)\rangle, k_{bs}, r')\rangle, k_{as}, r)\rangle \end{aligned}$$

Note that other processes should be added to considered corrupted agents or roles A, B and S talking to other agents but this would not really change the following sets of messages.

The output messages are:

$$\mathcal{M}_o = \left\{ \begin{array}{l} a, b, n_a \\ \pi_2(\pi_2(\pi_2(\mathrm{dec}(z_a, k_{as})))) \\ \mathrm{enc}(\langle\pi_2(\pi_2(z_s)), \pi_1(\pi_2(z_s)), \\ k_{ab}, \mathrm{enc}(\langle k_{ab}, \pi_1(z_s)\rangle, k_{bs}, r')\rangle, k_{as}, r) \end{array} \right\}$$

The tests are:

$$\left\{ \begin{array}{l} \pi_1(\mathrm{dec}(z_a, k_{as})) = n_a \\ \pi_1(\pi_2(\mathrm{dec}(z_a, k_{as}))) = b \end{array} \right\}$$

We define $\max\overline{\mathcal{E}_i} = \{\overline{e} \mid e \in \mathcal{E}_i\}$ in order to increase readability, and since it is easy to deduce $\overline{\mathcal{E}_i}$ from $\max\overline{\mathcal{E}_i}$.

$$\mathcal{D}_o = \{\pi_2(\pi_2(\pi_2(\mathrm{dec}(z, k_{as}))))\}$$

$$\mathcal{E}_0 = \{\mathrm{enc}(\langle z_1, \langle z_2, \langle \mathtt{x}, z_3\rangle\rangle\rangle, k_{as}, r), \mathrm{enc}(\langle \mathtt{x}, z_4\rangle, k_{bs}, r')\}$$

$$\max\overline{\mathcal{E}}_0 = \{\pi_1(\pi_2(\pi_2(\mathrm{dec}(z, k_{as})))), \pi_1(\mathrm{dec}(z, k_{bs}))\}$$

$$\mathcal{D}_o \cap \overline{\mathcal{E}}_0 = \emptyset$$

$$\mathcal{M}_t^{k_{ab}} = \emptyset$$

We deduce that P_{NS} is a well-formed process w.r.t. k_{ab}. Applying Theorem 2 and since the Needham-Schroeder symmetric key protocol preserves the syntactic secrecy of k_{ab}, we deduce that the protocol preserves the strong secrecy of k_{ab} that is

$$P_{NS}({}^{M}/_{k_{ab}}) \approx_l P_{NS}({}^{M'}/_{k_{ab}})$$

for any public terms M, M' w.r.t. $\mathrm{bn}(P_{NS})$.

C.2 Wide Mouthed Frog Protocol (modified)

The protocol is described below:

$$\begin{array}{l} A \Rightarrow B : N_a \\ B \Rightarrow S : \{N_a, A, K_{ab}\}_{K_{bs}} \\ S \Rightarrow A : \{N_a, B, K_{ab}\}_{K_{as}} \end{array}$$

Again, the target secret is K_{ab}.
The corresponding process is:

$$P_{NS}(k_{ab}) = \nu k_{as}.\nu k_{bs}.(!A)|(!S)|(!\nu k.B(k))|B(k_{ab})$$

where

$$\begin{array}{rl} A = & \nu n_a.\overline{c}\langle n_a\rangle.c(z_a).[\pi_1(\mathrm{dec}(z_a, k_{as})) = n_a] \\ B(x) = & c(z_b).\nu r.\overline{c}\langle \mathrm{enc}(\langle z_b, a, x\rangle, k_{bs}, r)\rangle \\ S = & c(z_s).[\pi_1(\pi_2(\mathrm{dec}(z_s, k_{bs}))) = a]. \\ & \quad \nu r'.\overline{c}\langle \mathrm{enc}(\langle \pi_1(\mathrm{dec}(z_s, k_{bs})), b, \pi_2(\pi_2(\mathrm{dec}(z_s, k_{bs})))\rangle, k_{as}, r')\rangle \end{array}$$

Note that other processes should be added to considered corrupted agents or roles A, B and S talking to other agents but this would not really change the following sets of messages.

The output messages are:

$$\mathcal{M}_o = \left\{ \begin{array}{l} n_a \\ \mathrm{enc}(\langle z_b, a, k_{ab}\rangle, k_{bs}, r) \\ \mathrm{enc}(\langle \pi_1(\mathrm{dec}(z_s, k_{bs})), b, \\ \pi_2(\pi_2(\mathrm{dec}(z_s, k_{bs})))\rangle, k_{as}, r') \end{array} \right\}$$

The tests are:

$$\left\{ \begin{array}{l} \pi_1(\mathrm{dec}(z_a, k_{as})) = n_a \\ \pi_1(\pi_2(\mathrm{dec}(z_s, k_{bs}))) = a \end{array} \right\}$$

$$\mathcal{D}_o = \{\pi_1(\mathrm{dec}(z, k_{bs})), \pi_2(\pi_2(\mathrm{dec}(z, k_{bs})))\}$$

$$\mathcal{E}_0 = \{\mathrm{enc}(\langle z_1, \langle z_2, \mathtt{x}\rangle, k_{bs}, r)\rangle\}$$

$$\max \overline{\mathcal{E}}_0 = \{\pi_2(\pi_2(\mathrm{dec}(z, k_{bs})))\}$$

$$\mathcal{E}_1 = \{\mathrm{enc}(\langle z_1, \langle z_2, \mathtt{x}\rangle, k_{as}, r)\rangle\}$$

$$\max \overline{\mathcal{E}}_1 = \{\pi_2(\pi_2(\mathrm{dec}(z, k_{as})))\}$$

$$\mathcal{D}_o \cap \overline{\mathcal{E}}_1 = \emptyset$$

$$\mathcal{M}_t^{k_{ab}} = \emptyset$$

We obtain similarly the same conclusion as for the previous protocol.

SANA - Security Analysis in Internet Traffic through Artificial Immune Systems

Michael Hilker[1] and Christoph Schommer[2]

[1] University of Luxembourg, Campus Kirchberg
1359, Luxembourg, 6, Rue Coudenhove-Kalergi, Luxembourg
michael.hilker@uni.lu
[2] University of Luxembourg, Campus Kirchberg
1359, Luxembourg, 6, Rue Coudenhove-Kalergi, Luxembourg
christoph.schommer@uni.lu

Abstract. The Attacks done by Viruses, Worms, Hackers, etc. are a Network Security-Problem in many Organisations. Current Intrusion Detection Systems have significant Disadvantages, e.g. the need of plenty of Computational Power or the Local Installation. Therefore, we introduce a novel Framework for Network Security which is called SANA. SANA contains an artificial Immune System with artificial Cells which perform certain Tasks in order to to support existing systems to better secure the Network against Intrusions. The Advantages of SANA are that it is efficient, adaptive, autonomous, and massively-distributed. In this Article, we describe the Architecture of the artificial Immune System and the Functionality of the Components. We explain briefly the Implementation and discuss Results.

Keywords. Artificial Immune Systems, Network Security, Intrusion Detection, Artificial Cell Communication, Biological-Inspired Computing, Complex Adaptive Systems

1 Introduction

Companies, Universities, and other Organisations use connected Computers, Servers, etc. for Working, Storing of important Data, and Communication. These Networks are an Aim for Attackers in order to breakdown the Network Service or to gain internal and secret Information.

These Attacks are Intrusions which are e.g. Worms, Viruses, Hacker-Attacks. Network Administrators try to secure the Network against these Intrusions using Intrusion Detection Systems (IDS). The Network Intrusion Detection Systems (NIDS) are a local System which is installed in one important Node and which checks all Packets routed over this Node, e.g. SNORT [1] or [2,3,4,5,6]. Host-based Intrusion Detection Systems (HIDS) are installed on each Node and check each Packet which is routed over this Node [7,8,9]. Furthermore, there are approaches of distributed Intrusion Detection Systems (D-IDS) which install IDS on all machines and connect these; one example is SNORTNET [10].

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/694

Unfortunately, these IDS have several Disadvantages as for example the plenty of Computational Power, the need of Administration during Execution, and local Installation. Additionally, the Intrusions are getting both more and more complex and intelligent, so that the IDS have lots of Problems to identify the Intrusions, e.g. Camouflage of Attacks. Thus, novel Approaches for Network Security are needed which should provide the following features:

- Distributed: all Nodes should be secured and there should not be any central Center
- Autonomous: the System and all Components should work autonomously; hereby, the number of false-positives should be low
- Adaptive: the System should have the ability to identify or react to modified or even novel Attacks
- Cooperative: The Computational Power should be shared over the whole Network

In SANA, we introduce an artificial Immune System which provides the features explained above. In the next Section, we discuss existing artificial Immune Systems for the Application of Network Security.

2 Current Situation

For the explanation of the different existing artificial Immune Systems for Network Security, we will introduce briefly the Paradigm of artificial Immune Systems [11]:

An artificial Immune System tries to simulate the human Immune System which secures the Human Body against Pathogens [12]. An artificial Immune System is a massively distributed System and Complex Adaptive System with lots of components. In the human Immune System, these Components are e.g. Cells, Lymph-Nodes, Bone Marrow. All of these Components work autonomously, efficiently and are highly specialised. These Components cooperate using the Cell Communication with e.g. Cytokines and Hormones. Additionally, there are lots of cellular and immunological Processes which mesh in the Protection of the Human Body. The artificial Immune Systems try to model these. Unfortunately, the human Immune System and the Modelling of it is so complex and partly not understood. Therefore, artificial Immune Systems can only model a part of the human Immune System.

There are several artificial Immune Systems for Network Security. We discuss some interesting Approaches of artificial Immune Systems for Network Security:

Spafford and Zamboni introduce in [13] a System for Intrusion Detection using autonomous Agents. These Agents cooperate with Transceivers and do not move through the Network. Hofmeyr and Forrester [14,15,16] introduce an artificial Immune System for Network Security (named ARTIS/LISYS). The AIS models the Lifecycle of T- and B-Cells with positive and negative Selection. The non-mobile Detectors check a Triple of Source-IP, Destination-IP and Destination-Port and evaluate if a Packet is malicious or not. Additionally, in

this Broadcast-Network, all Detectors see all Packets and react to it. In [17] an artificial Immune System as a Multi-Agent System is introduced for Intrusion Detection. The system uses mobile Agents which cooperate with a centralised Database containing the Attack-Information.

In the next Section we introduce the Architecture of the artificial Immune System SANA. In contrast to the existing artificial Immune Systems, SANA uses autonomous, fully-mobile, and lightweighted artificial Cells; additionally, SANA does not have any centralised System. Furthermore, SANA is not a closed Framework; it is possible to use existing Network Security Approaches in SANA. Thereafter, we take a closer look on the different Components of the artificial Immune System.

3 SANA - Architecture

The artificial Immune System of SANA secures the whole Network against Intrusions and provides the Features explained above. In SANA, we simulate a packet-oriented Network using a Network Simulator (see Section 3.1). SANA is a collection of non-standard Approaches for Network Security and we test if they increase the Performance of existing Network Security Systems. An Adversarial injects Packets with and without Attacks in order to stress the Network and the artificial Immune System as well as to simulate Attacks (Section 3.2).

The artificial Immune System uses several Components for the Security of the Network. All of these Components work autonomously and there is no Center which is required by any Component. The main Components are artificial Cells, Packet-Filters, IDS, etc. Packet-Filters are a local System that check the Header of each Packet. IDS are local, non-mobile Systems which check Packets and observe the Network Traffic in order to secure the Node where the IDS is installed. Artificial Cells (Section 3.3) are autonomous, fully-mobile, and lightweighted Entities which flow through the Network and perform certain Tasks for Network Security, e.g. Packet-Checking, Identification, of Infected Nodes or Monitoring of the Network. Furthermore, artificial Cell Communication (Section 3.4) is used to initialise Cooperation and Collaboration between the artificial Cells and a Self-Management (Section 3.5) is utilised for a Regulation of the artificial Immune System. In the next Sections, we take a closer look on the different Components of SANA.

3.1 Network Simulator, Security Framework and Workflow

The Network Simulator simulates a Packet-Oriented Network and is based on the Adversarial Queueing Theory [18,19,20]. The Simulator uses a FIFO (First In First Out) approach for Queueing and for Routing the Shortest Path Routing with the Dijkstra-Algorithm. It has a Quality of Service (QoS) Management which prefers artificial Cells and other important Messages that are sent between certified Components of the AIS.

The Security Framework is the AIS which must be installed on each Node of the Network. Furthermore, this Framework guarantees e.g. the execution of the artificial Cells, the Presentation of Packets to all Security Components, the Sending of Messages. The Design of the Security Framework is focussed on Expandability in order to enhance it and to use existing Approaches in Network Security. One example of a Network Security Approach is Malfor [21], a system for Identification of the Processes which are involved in the Installation of an Intrusion.

The Workflow is that each Packet is checked in each Node by every Security Component - e.g. artificial Cells, Packet-Filters, and IDS - each Security Component can perform other Tasks - e.g. moving to other Nodes or sending Messages - and the Adversarial injects Packets into the Network.

3.2 Adversarial and Attacks

An Adversarial has the Function to Stress the Network and the AIS using Packets with and without Attacks; it has to keep in mind that the bandwidth of the connection is limited and that the queues have limited size. The Adversarial injects Packets without Attacks in order to simulate a real Network. The Packets with Attacks try to infect Nodes with Attacks; the infected Nodes then perform certain Tasks depending on the Attack, e.g. sending Packets with Attack to other Nodes. The Attack is an abstract Definition for all Intrusions in SANA. So, nearly all Intrusions can be modelled, e.g. Worms, Viruses, and Hacker-Attacks.

3.3 Artificial Cells

Artificial Cells are the main Component in the artificial Immune System of SANA. An artificial Cell is a highly specialised, autonomous and efficient Entity which flows through the Network and performs certain Tasks for Network Security. In the Cooperation and with the enormous Number of artificial Cells, the whole System adapts quickly to Attacks and even to modified and novel Attacks; the idea of Complex Adaptive Systems (CAS) or Massively-Distributed Systems.

Each artificial Cell has the Job to perform some certain Task:

- ANIMA for Intrusion Detection which is a type of artificial Cells for checking Packets whether they contain an Attack or not. Furthermore, it compresses the Information how to identify and how to proceed if an Attack is found in order to save Storage-Space and Computational Power. More Information about ANIMA-ID can be found in [22].
- AGNOSCO which is a type of artificial Cells for the Identification of Infected Nodes using artificial Ant Colonies. It is a distributed System which identifies the infected Nodes quickly and properly. More Information can be found in [23].
- Monitoring artificial Cell which flows through the Network and collects Information about the Status and send this back to some certain Component, e.g. the Administrator.

– Using the Expandability of SANA, it is easily possible to introduce novel artificial Cells. Thus, it is e.g. possible to introduce artificial Cells for Anomaly Detection or Checking of the Status of a Network Node.
– Additionally, it is possible to use existing Approaches for Network Security. With the Expandability of SANA, these Approaches can be used in an artificial Cell; examples are Systems for Intrusion- [22,24] or Anomaly-Detection Systems [25,26,27].

3.4 Artificial Cell Communication

The idea in Complex Adaptive System (CAS) is that the Components (here: artificial Cells) perform basic Tasks, are highly specialised and use basic Systems for Cooperation. Only by Cooperation and the high amount of these Components, the System is adaptive and reaches the goal (here: Network Security).

The whole Architecture in SANA is composed without any central System. Thus, the artificial Cell Communication cannot use a Central Management System like it is used in several Multi Agent Systems or Ad-Hoc Networks. We model partly the Cell Communication of the Human Body in order to build up Communication and, thereafter, Cooperation between artificial Cells.

We introduce the Term Receptor which is a Public-Key-Pair. Each Component has Receptors and each Message is packed into a Substance which is an encrypted Message with Receptors. Only if a Receiver has the right Set of Receptors, it will receive the Message - the Idea of a Public-Key Infrastructure and widely used in Multi Agent System for the Disarming of Bad-Agents/-artificial Cells; however, in our Implementation, there is not any centralised Key-Server.

Additionally, we introduce artificial Lymph Nodes and Central Nativity and Training Stations (CNTS). Artificial Lymph Nodes supply the artificial Cells with e.g. Knowledge, initiate other artificial Cells if an event occurs and artificial Lymph Nodes care about the Routing of Substances. CNTS train and release new artificial Cell in order to have an evolutionary Set of artificial Cells which are up-to-date. Both, artificial Lymph Nodes and Central Nativity and Training Stations, are redundant installed in the System.

3.5 Self-Management of the artificial Immune System

The Self-Management of the System is currently only rudimentary. The artificial Cells are autonomous and thus they flow through the Network and perform certain Tasks. However, one Problem of Massively-Distributed Systems or Complex Adaptive Systems is that they just do their Tasks but there is not any guarantee that the Systems will do the Tasks successfully. On the basis of the artificial Cell Communication and novel Structures, we want to introduce a distributed Self-Management of the artificial Immune System in order to give a certain amount of Guarantee. However, this is one of the Next Steps explained in the Section 6.

4 SANA - Implementation

The Project SANA is implemented in Java. The Network Simulator, Adversarial, and the artificial Immune System are implemented and running. Different Types of artificial Cells are implemented. The Performance of these artificial Cells is tested and they perform the Tasks properly. Attack-Scenarios are additionally implemented for Testing Purposes and one example is a realistic Worm-Attack which will be discussed in the Section 5.1.

The whole Implementation has the aim to give a Prototype for Testing and Evaluation of the Approaches. Furthermore, the Implementation focuses more on Expandability than on Performance; it is also possible to model nearly all Intrusions and nearly all immunological Processes. It is also possible to add common used Network Security Solutions like SNORT [1] or Malfor [21]. With this, we can compare the Performance of SANA with common used IDS and we can model cooperation between SANA and IDS.

5 SANA - Results

The Results we gained are promising. SANA identifies most Attacks - about 60%-85% - depending on the Attack-Behaviour, the Network Topology and the Behaviour of the artificial Immune System with the artificial Cells. The infected Nodes are identified quickly by AGNOSCO and the System adapts to Attacks using local Immunization.

If there are IDS or especially NIDS in the Network which protect important Nodes like the Internet Gateway or the E-Mail-Server, there is cooperation between SANA and the IDS with a good performance - about 80%-95% of the Attack are prevented. Thus, SANA does not replace existing IDS, it enhances them.

In the next Section, we discuss the Results of a Simulation of a realistic Worm-Attack.

5.1 Simulation of a Worm-Attack

In this Section, we discuss a Modelling of a realistic Worm-Attack onto the Network. The Worm enters a Network and uses a Security-Hole in a Node in order to install itself. After this, the Worm tries to propagate it to other Nodes; therefore, it sends lots of Packets containing a copy of it to other Nodes. SANA tries to identify and remove these Packets, identifies the infected Nodes and disinfects the identified infected Nodes. Therefore, SANA uses the different types of artificial Cells explained in the Section 3.3 and the artificial Cell Communication explained in the Section 3.4.

The Performance of SANA in this Simulation is promising. It secures other Nodes from being infected by this Worm using ANIMA for Intrusion Detection [22]; only some Neighbour-Nodes are infected (about 2-5 Nodes for each Infection). It also identifies the infected Nodes using AGNOSCO [23] quickly (about

50-150 Time-Steps for each infected Node) and using the artificial Cell Communication (Section 3.4), AGNOSCO informs the artificial Lymph-Nodes (Section 3.4) which start an artificial Cell for Disinfection which disinfect the Node fast. To sum up, SANA protects the Network against a Worm-Attack properly.

5.2 Theoretical Analysis of distributed IDS

In the theoretical Part of the SANA-Project, we compare the Performance and the Need of Resource of distributed and centralised Network Security Systems. Examples for centralised are e.g. IDS and for distributed AIS. However, the Analysis shows quickly that the Performance of the both Approaches is highly dependent on the Network Topology and the Behaviour of the Intrusions. The Analysis fortunately shows that the Performance of IDS is increased if AIS are added and the additionally needed Resources are limited.

6 SANA - Next Steps

Next Steps in the SANA-Project are to simulate realistic Attacks on Networks, e.g. different Worm, Virus and Malwar-Attacks; also Attacks which consists of several different Attacks. Additionally, another part is to increase the Performance of the artificial Cell Communication (Section 3.4) and analyse the Performance of it theoretically. Furthmore, we will introduce a Self-Management (Section 3.5) which guarantees a certain amount of Security and we will perform further theoretical Comparison (Section 5.2) between distributed and centralised Network Security Systems.

7 Conclusion

Network Security is still a challenging field. Unfortunately, the Attacks are getting both more complex and intelligent. Therefore, existing Network Security Systems have problems to cope with these Problems. We introduce with SANA an artificial Immune System with several non-standard Approaches for Network Security. With the gained Results, we are sure that SANA will enhance current Network Security Systems.

One last word about SANA: SANA is Latin and stands for healthy. Furthermore, the Work is done interdisciplinary in cooperation between Researchers from Biology and Computer Science.

Acknowledgments

The PhD-Project SANA is part of the project INTRA (= INternet TRAffic management and analysis) that are financially supported by the University of Luxembourg. We would like to thank the Ministre Luxembourgeois de l'education et de la recherche for additional financial support.

References

1. Roesch, M.: Snort - lightweight intrusion detection for networks. LISA **13** (1999) 229–238
2. Debar, H., Dacier, M., Wespi, A.: Towards a taxonomy of intrusion-detection systems. Computer Networks **31** (1998) 805–822
3. Snapp, S.R., Brentano, J., Dias, G.V., Goan, T.L., Heberlein, L.T., lin Ho, C., Levitt, K.N., Mukherjee, B., Smaha, S.E., Grance, T., Teal, D.M., Mansur, D.: DIDS (distributed intrusion detection system) - motivation, architecture, and an early prototype. National Computer Security Conference **14** (1991) 167–176
4. Staniford-Chen, S., Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagland, J., Levitt, K., Wee, C., Yip, R., Zerkle, D.: Grids - a graph based intrusion detection system for large networks. National Information Systems Security Conference **19** (1996)
5. Janakiraman, R., Waldvogel, M., Zhang, Q.: Indra: A peer-to-peer approach to network intrusion detection and prevention. Proceedings of IEEE WETICE 2003 (2003)
6. Antonatos, S., Anagnostakis, K., Polychronakis, M., Markatos, E.: Performance analysis of content matching intrusion detection systems. SAINT **4** (2004)
7. Wagner, D., Dean, D.: Intrusion detection via static analysis. In IEEE Symposium on Security and Privacy (2001)
8. Lindqvist, U., Porras, P.A.: expert-bsm: A host-based intrusion detection solution for sun solaris. In Proceedings of the 17th Annual Computer Security Applications Conference (2001) 240–251
9. Chari, S.N., Cheng, P.C.: Bluebox: A policy-driven, host-based intrusion detection system. ACM Transactions on Information and System Security **6** (2003) 173–200
10. Fyodor, Y.: Snortnet' - a distributed intrusion detection system. [Online]. Available: http://snortnet.scorpions.net/snortnet.pdf (2000)
11. DeCastro, L.N.: Artificial Immune Systems: A New Computational Intelligence Approach. First edn. Springer (2002)
12. Janeway, C.A., Travers, P., Walport, M., Shlomchik, M.: Immunobiology: the Immune System in Health and Disease. Sixth edn. Garland Publishing (2004)
13. Spafford, E.H., Zamboni, D.: Intrusion detection using autonomous agents. Computer Networks **34** (2000) 547–570
14. Hofmeyr, S.A., Forrest, S.: Immunity by design: An artificial immune system. Proceedings of the Genetic and Evolutionary Computation Conference **2** (1999) 1289–1296
15. Hofmeyr, S.A., Forrest, S.: Architecture for an artificial immune system. Evolutionary Computation **8** (2000) 443–473
16. Hofmeyr, S.A., Forrest, S.: Immunology as information processing. (2000)
17. Machado, R.B., Boukerche, A., Sobral, J.B.M., Juca, K.R.L., Notare, M.S.M.A.: A hybrid artificial immune and mobile agent intrusion detection based model for computer network operations. IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 6 **19** (2005)
18. Andrews, Baruch Awerbuch, Antonio Fernndez, Tom Leighton, Zhiyoung Liu and Jon Kleinberg, M.: Universal-Stability Results and Performance Bounds for Greedy Contention-Resolution Protocols. Journal of the ACM **48** (2000) 39–69
19. Hilker, M.: Queueing Strategies in Internet Routing. Diploma Thesis at the Johann Wolfgang Goethe-University Frankfurt/M., Germany (2005)

20. Hilker, M., Schommer, C.: A new queueing strategy for the adversarial queueing theory. IPSI-2005 Slovenia (2005)
21. Neuhaus, S., Zeller, A.: Isolating intrusions by automatic experiments. 13th Annual Network and Distributed System Security Symposium (2006)
22. Hilker, M., Schommer, C.: Description of bad-signatures for network intrusion detection. AISW-NetSec 2006 during ACSW 2006, CRPIT **54** (2006)
23. Hilker, M., Schommer, C.: Agnosco - identification of infected nodes with artificial ant colonies. RASC 2006 (2006)
24. Finizio, I., Mazzariello, C., Sansone, C.: A temporal-behavior knowledge space for detecting intrusions in computer networks. RASC 2006 (2006)
25. Sekar, R., Gupta, A., Frullo, J., Shanbhag, T., Tiwari, A., Yang, H., Zhou, S.: Specification-based anomaly detection: a new approach for detecting network intrusions. Volume 9. (2002) 265–274
26. Lazarevic, A., Ertoz, L., Ozgur, A., Srivastava, J., Kumar, V.: A comparative study of anomaly detection schemes in network intrusion detection. Proceedings of Third SIAM Conference on Data Mining **3** (2003)
27. Leung, K., Leckie, C.: Unsupervised anomaly detection in network intrusion detection using clusters. Australasian Computer Science Conference **28** (2005)

Shape Analysis of Sets

Jan Reineke

Saarland University
Im Stadtwald - Gebäude E1 3
66041 Saarbrücken, Germany
reineke@cs.uni-sb.de

Abstract. Shape Analysis is concerned with determining "shape invariants", i.e. structural properties of the heap, for programs that manipulate pointers and heap-allocated storage. Recently, very precise shape analysis algorithms have been developed that are able to prove the partial correctness of heap-manipulating programs. We explore the use of shape analysis to analyze abstract data types (ADTs). The ADT Set shall serve as an example, as it is widely used and can be found in most of the major data type libraries, like STL, the Java API, or LEDA. We formalize our notion of the ADT Set by algebraic specification. Two prototypical C set implementations are presented, one based on lists, the other on trees. We instantiate a parametric shape analysis framework to generate analyses that are able to prove the compliance of the two implementations to their specification.

1 Introduction

This paper deals with the Shape Analysis of the Abstract Data Type (ADT) Set. Its main goal is to use Shape Analysis to prove that Set implementations written in C comply to an algebraic specification of the ADT Set. The paper summarizes major results from the author's Master's thesis [Rei05].

Shape Analysis [CWZ90,GH96,SRW99,SRW02] is concerned with determining "shape invariants", i.e. structural properties of the heap, for programs that manipulate pointers and heap-allocated storage. Formerly, it was primarily used to aid compilers. Knowledge about the structure of the heap allows to carry out several optimizations, for instance, compile-time garbage collection, better instruction scheduling and automatic parallelization.

Recently, more precise shape analysis algorithms have been developed that are able to prove the partial correctness of heap-manipulating programs. In [LARSW00] bubble-sort and insertion-sort procedures are analyzed. The analyses were able to infer that the procedures indeed returned sorted lists. They also successfully analyzed destructive list reversal and the merging of two sorted lists. The analyses of [LARSW00] and our analyses are based on the Shape Analysis Framework presented in [SRW02]. Logical structures are used to represent the program state in this framework. The concrete semantics is specified in first-order logic. By interpreting the concrete semantics in a 3-valued domain sound and precise abstractions can be extracted automatically.

Set implementations are widely used and can be found in most of the major data type libraries, like STL [MS96], the Java API [Mic04], or LEDA [MN99]. The ADT Set shall serve as an example of abstract data types. The main goal of this paper is to show the partial correctness of set implementations using Shape Analysis. For this purpose we formally define the ADT Set using algebraic specification [EM85,EM90,LEW97]. It shall serve as a reference for the implementations described later. Algebraic Specification allows us to express the intended behaviour independently of possible concrete implementations. The following two axioms are taken from our definition:

$$a \in s.\texttt{insert}(b) \leftrightarrow a =_{el} b \vee a \in s, \quad (3)$$
$$a \in s.\texttt{remove}(b) \leftrightarrow a \neq_{el} b \wedge a \in s \quad (4)$$

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/698

They capture the effect of the $\cdot$.`insert`($\cdot$)- and $\cdot$.`remove`($\cdot$)-functions on the $\in$-predicate. Notice that they do not make any statement about the concrete data structures or algorithms employed.

We present two prototypical C implementations, one based on singly-linked lists, the other on binary trees. Using Shape Analysis, we demonstrate that these implementations comply to our specification of the data type. This involves creating precise analyses using the framework of [SRW02] and linking the results to the specification of the ADT.

2 Sets as Data Abstractions

The formal definition of the ADT Set will serve as a reference for the implementations introduced later. The definition should be independent of possible implementations. Notice that a concrete implementation would also constitute a formal specification. It would however contain many design decisions that are not specific to the data type itself.

A method widely used for the specification of data types is known as *Algebraic Specification of Data Types* [EM85,EM90,LEW97]. Here, a specification consists of a signature and axioms. The signature introduces operations on the data type, while the axioms capture the meaning of the given operations. Data Types defined in this way are often called Abstract Data Types. This is for three reasons:

- The specification is concerned with the data type itself as an abstract mathematical object and not with its implementation by a concrete program in a particular programming language.
- Specifications may be incomplete by only partially specifying the meaning of operations.
- They maybe defined in terms of other data types that serve as parameters. This is also called generic specification.

While we easily grasp an intuitive meaning of these specifications, it is of course profitable to give a formalization of the concept. We will not go into detail about this since we do not rely on the precise definitions in the following chapters. The semantics of such a specification is a set of many-sorted algebras. An algebra belongs to this set if it is a model of the axioms of the specification. The axioms are implicitly universally quantified. Usually, there are many non-isomorphic models of a given specification reflecting the incompleteness of the definition. The interested reader may consult [EM85] and [LEW97] for an in-depth treatment of the topic.

The full specification of the ADT Set is displayed in Table 1. Our specification is parameterized by an *element* type. This could also be instantiated with a *set* itself, building sets of sets of some primitive type, and so on. We are assuming an existing specification of the natural numbers *nat*.

The empty set is provided as a constant. Other sets can be constructed by inserting and removing elements using `.insert`($\cdot$) and `.remove`($\cdot$). The `.selectAndRemove` function returns an element and removes it from the set. It can be used to iterate over a set. The `.sizeOf` function returns the cardinality of the set as a natural number. The $\in$ predicate allows to test set membership. $\subseteq$ and $=$ correspond to subset and equality of sets.

Most of the axioms are straightforward. We distinguish equality on sets $=$, equality on elements $=_{el}$, and equality on natural numbers $=_{nat}$. Axiom (1) assures that every possible set can be constructed by applications of $\varnothing$ and `.insert`. In axiom (5) we only have an implication because the `.selectAndRemove` function chooses an element nondeterministically. Axioms (6) and (7) correspond to the extensionality axiom of set theory. Axioms (8)-(13) deal with the cardinality of sets. The axioms are complete in the sense that the meaning of arbitrary formulae over the given alphabet (the functions and predicates of the ADT specification) can be derived.

set =
begin generic specification

parameter	element					
using	nat					
sorts	set					
constants	$\varnothing$	:	set			
functions	$\cdot$.`insert`($\cdot$)	:	set	$\times$ element	$\rightarrow$	set
	$\cdot$.`remove`($\cdot$)	:	set	$\times$ element	$\rightarrow$	set
	$\cdot$.`selectAndRemove`	:	set		$\rightharpoonup$	element $\times$ set
	$\cdot$.`sizeOf`	:	set		$\rightarrow$	nat
predicates	$\cdot \in \cdot$	:	element	$\times$ set		
	$\cdot \subseteq \cdot$	:	set	$\times$ set		
	$\cdot = \cdot$	:	set	$\times$ set		
variables	s, s'	:	set			
	a, b	:	element			

axioms set **generated by** $\varnothing$, `.insert`; (1)

$\neg(a \in \varnothing)$, (2)

$a \in s.\texttt{insert}(b) \leftrightarrow a =_{el} b \vee a \in s$, (3)

$a \in s.\texttt{remove}(b) \leftrightarrow a \neq_{el} b \wedge a \in s$, (4)

$(a, s') = s.\texttt{selectAndRemove} \rightarrow a \in s \wedge a \notin s' \wedge s'.\texttt{insert}(a) = s$, (5)

$s \subseteq s' \leftrightarrow a \in s \rightarrow a \in s'$, (6)

$s = s' \leftrightarrow s \subseteq s' \wedge s' \subseteq s$, (7)

$\varnothing.\texttt{sizeOf} =_{nat} 0$, (8)

$s.\texttt{insert}(b).\texttt{sizeOf} =_{nat} s.\texttt{sizeOf} \leftrightarrow b \in s$, (9)

$s.\texttt{insert}(b).\texttt{sizeOf} =_{nat} s.\texttt{sizeOf} + 1 \leftrightarrow \neg(b \in s)$, (10)

$s.\texttt{remove}(b).\texttt{sizeOf} =_{nat} s.\texttt{sizeOf} \leftrightarrow \neg(b \in s)$, (11)

$s.\texttt{remove}(b).\texttt{sizeOf} =_{nat} s.\texttt{sizeOf} - 1 \leftrightarrow b \in s$, (12)

$(a, s') = s.\texttt{selectAndRemove} \rightarrow s'.\texttt{sizeOf} =_{nat} s.\texttt{sizeOf} - 1$. (13)

end generic specification

Table 1. ADT Set

3 Shape Analysis of Implementations

In this section we analyze two prototypical C implementations of the ADT Set. One implementation is based on singly-linked lists, the other on binary trees. After briefly introducing parts of the two implementations, we proceed to describe our analyses. The main goal of the analyses is to prove that the implementations comply with the ADT specification given in Chapter 2. The implementations each contain the two methods, `insertElement`, `removeElement` and the function `isElement`. They implement the $\cdot$.`insert`($\cdot$), $\cdot$.`remove`($\cdot$) functions and the $\cdot \in \cdot$ predicate, respectively. We chose to show the following two axioms, since they capture the most important aspects of the ADT Set:

$$a \in s.\texttt{insert}(b) \leftrightarrow a =_{el} b \vee a \in s, \quad (3)$$
$$a \in s.\texttt{remove}(b) \leftrightarrow a \neq_{el} b \wedge a \in s \quad (4)$$

Our analyses are conducted using TVLA [LAS00] and are based on previous analyses on lists and trees contained in the TVLA 2 distribution.

3.1 List-based Implementation

```
typedef struct List
{
  void* data;
  struct List* next;
} List;

typedef struct Set
{
  List* list;
  int (*compare)(void*, void*);
  int size;
} Set;
```

(a)

```
int isElement(Set* set, void* element)
{
  List* list = set->list;

  while (list != 0)
  {
    if (compare(list->data, element) == 0)
      return 1;

    list = list->next;
  }

  return 0;
}
```

(b)

Fig. 1. C structure declarations for Lists and Sets and C source of membership test

Our first set implementation uses singly-linked lists to store the elements. It also maintains the size of the current set. The structure declarations are visible in Figure 1. When allocating such a set, a compare-function has to be given, that establishes an equivalence relation on the data elements.

Figure 1 also shows the code for testing set membership. The method simply iterates over the list, comparing each item with the element that is tested for set membership.

Figure 2 shows the implementations of the insertion and removal methods. The insertion method iterates over the list until it either finds the element or reaches the final element of the list, indicated by a null-pointer in the next-field. If the element was not found it is appended at the end. Removal works similarly. When the element is found, it is decoupled from the list and the memory is freed.

Data Structure Invariants Our analyses rely on a number of data structure invariants at entrance to the methods. Showing their maintenance is part of the proof. By data structure invariants we mean invariants that are related directly to the concrete data structure employed to implement the ADT Set. In this case properties of singly-linked lists:

– The list is acyclic
– The list does not contain any duplicate elements

We use instrumentation predicates to capture these properties formally using first-order logic.

3.2 Tree-based Implementation

As in the list-based case, a compare-function is needed. This time it has to implement a reflexive total order. This is necessary, to build an ordered tree. Figure 3 shows the structure declarations. Every node in the tree stores one of the set elements and maintains pointers to two children nodes *left* and *right*.

Figure 3 also contains the source of the set membership test. The method simply traverses the tree until it either finds the element or reaches a leaf node. The source of the insertion and removal methods on trees can be found in the appendix, since it is too large to be dealt with here. We restrict ourselves to mentioning the main ideas of the two algorithms. New elements are always inserted as new leaf nodes, by traversing the tree to the correct position. While insertion of elements if fairly easy and quite similar to its list pendant, removal of elements is a non-trivial task. Figure 4 illustrates this. Removing elements that are stored in leaf nodes is simple (left). They can simply be decoupled from their respective parent nodes. If the node has one child, we can connect this child at the place of the node to its former parent node (middle). The most complicated case arises when the particular node has two child nodes (right). In this case, we have to find another node in the tree to replace the element node. This node has to be smaller than all nodes on the right and greater than all nodes on the left. There are two ways to find such an element. Either one can take the right-most element of the left subtree or the left-most element of the right subtree. We chose to always take the right-most element of the left subtree. In addition, there are some special cases of the latter case. For instance, if the root of the left subtree is already the right-most element of the left subtree.

```
void insertElement(Set* set, void* element)
{
  List* list = set->list;
  List* prev = 0;

  while (list != 0)
  {
    if (compare(list->data, element) == 0)
        return;

    prev = list;
    list = list->next;
  }

  List* newList = (List*)malloc(sizeof(List));
  newList->data = element;
  newList->next = 0;
  set->size++;

  if (prev == 0) //list is empty
  {
    set->list = newList;
  }
  else //append item to list
  {
    prev->next = newList;
  }
}
```

(a)

```
void* removeElement(Set* set, void* element)
{
  List* temp;
  List* list = set->list;

  if (list == 0)
    return;

  if (compare(list->data, element) == 0)
  {
    set->size--;
    set->list = list->next;
    free(list);
  }
  else
    while (list->next != 0)
    {
      if (compare(list->next->data, element) == 0)
      {
        void* deletedElement = list->next->data;
        set->size--;
        temp = list->next->next;
        free(list->next);
        list->next = temp;
        return deletedElement;
      }
      list = list->next;
    }
}
```

(b)

Fig. 2. C source of Insertion and Removal methods

```
typedef struct Tree
{
  void* data;
  struct Tree* left;
  struct Tree* right;
} Tree;

typedef struct Set
{
  Tree* tree;
  int (*compare)(void*, void*);
  int size;
} Set;
```

(a)

```
int isElement(Set* set, void* element)
{
  Tree* tree = set->tree;

  while (tree != 0)
  {
    if (compare(tree->data, element) == 0)
      return 1;
    else if (compare(tree->data, element) < 0)
      tree = tree->left;
    else
      tree = tree->right;
  }

  return 0;
}
```

(b)

Fig. 3. C structure declarations for Trees and Sets and C source of isElement test

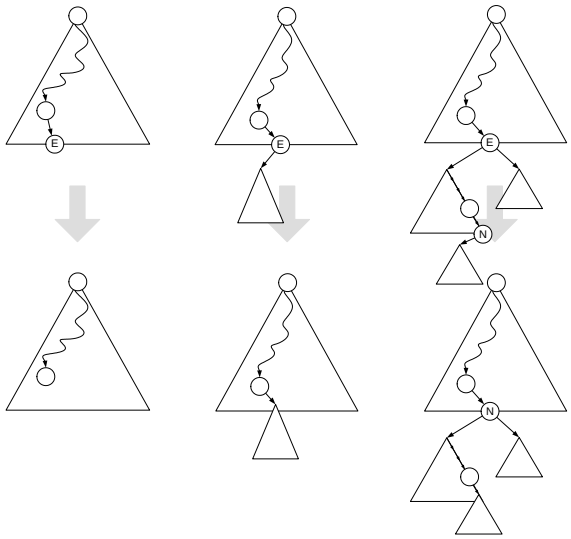


Fig. 4. Removal from Ordered Tree

Data Structure Invariants In order to prove our ADT Set axioms we need to maintain two data structure invariants:

- The structure representing the set is a tree
  Out of many equivalent definitions for "binary treeness", we chose the following: Whenever an element is reachable from the left child of a node in the structure, then it is not reachable from the right child, and vice versa.
- The tree is ordered
  Every element reachable from the left child is smaller and every element reachable from the right child is greater. This implies that the tree does not contain duplicate elements. It also implies the first data structure invariant. It is still useful to consider the first invariant, because it may help in proving this one.

Again, we used instrumentation predicates to formalize the two invariants using first-order logic. Proving the latter proved to be quite difficult. It is a global property, i.e. it does relate elements in the tree that are not directly connected. We will go into more detail about this in the analysis section.

3.3 Shape Analysis

To prove the ADT Set axioms we perform three analyses for each implementation. The analyses of the insertion methods prove the following:

$$isElement(a, s.\texttt{insertElement}(b)) \leftrightarrow a =_{el} b \vee isElement(a, s)$$

Notice the difference compared with the corresponding axiom (3). The instrumentation predicate *isElement* replaces the $\cdot \in \cdot$ predicate. That is we prove the property of the insertion method in terms of an instrumentation predicate. The same holds for the removal methods and axiom (4). There, we prove:

$$isElement(a, s.\texttt{removeElement}(b)) \leftrightarrow a \neq_{el} b \wedge isElement(a, s)$$

To conclude the proofs we show that the `isElement` functions in both implementations are equivalent to the instrumentation predicate *isElement*:

$$isElement(a, s) \leftrightarrow s.\texttt{isElement}(a)$$

Combining this equivalence with the two preceding proofs yields:

$$s.\texttt{insertElement}(b).\texttt{isElement}(a) \leftrightarrow a =_{el} b \vee s.\texttt{isElement}(a)$$
$$s.\texttt{removeElement}(b).\texttt{isElement}(a) \leftrightarrow a \neq_{el} b \wedge s.\texttt{isElement}(a)$$

These two equivalences correspond directly to axioms (3) and (4).

Shape Analysis of List-based Implementation Our analysis is based on existing analyses on lists and trees. We borrowed the concrete semantics of most of the statements from these. The following table shows how we represent the state by logical predicates.

Predicate	**Intended Meaning**
$x(v)$ for each $x \in Var$	Pointer variable x points to heap cell v.
$n(v_1, v_2)$	The *next* selector of v_1 points to v_2.
$deq(v_1, v_2)$	The *data*-fields of v_1 and v_2 are equal.
$isSet(v)$	v represents a set.
$or[n, x](v)$ for each $x \in Var$	v was reachable from x via *next*-fields.

As depicted, pointer variables are represented by unary predicates. The *next*-field is modeled by a binary predicate. Since we can only model the structure of the heap by these predicates, primitive values have to be dealt with differently. Abstracting from the concrete values of the *data*-fields, we capture the equivalence relation between *data*-fields by the binary predicate *deq*. This corresponds to the compare-function needed in the implementation. To differentiate between set locations and other locations in the heap, the *isSet* predicate is used. To be able to relate elements contained in the list before the execution of one of our procedures with their output structures, we mark elements reachable from x via *next*-fields using the $or[n, x]$ predicate.

While the above core predicates suffice to define the concrete semantics of all the statements, we need additional instrumentation predicates to gain precision.

Predicate	**Defining Formula**	**Intended Meaning**
$is[n](v)$	$\exists v_1, v_2.(v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v))$	v is shared.
$c[n](v)$	$\exists v_1.(n(v_1, v) \wedge n^*(v_1, v_2))$	v resides on a cycle.
$t[n](v_1, v_2)$	$n^*(v_1, v_2)$	Transitive reflexive closure of *next*.
$r[n, x](v)$ for each $x \in Var$	$\exists v_1.(x(v_1) \wedge t[n][v_1, v))$	v is reachable from x via *next*-fields.
$noeq[deq, n](v)$	$\forall v_1.(((t[n](v_1, v) \vee t[n](v, v_1)) \wedge v_1 \neq v) \rightarrow (\neg deq(v_1, v) \wedge \neg deq(v, v_1)))$	The *data*-field of v is different from the *data*-fields of locations that can reach v and that are reachable from v.
$validSet(v)$	$isSet(v) \wedge noeq[deq, n](v)$	v represents a valid set (no duplicate entries).
$isElement(v_1, v_2)$	$isSet(v_2) \wedge \exists v.(t[n](v_2, v) \wedge deq(v_1, v) \wedge v \neq v_2)$	v_1 is an element of set v_2.

The first four of these instrumentation predicates capture general properties of the shape of the heap. They have been used in previous analyses of list-manipulating programs. $c[n]$ covers the acyclicity data structure invariant mentioned in the implementation section.

The $noeq[deq, n]$ predicate is tailored specifically to the current task. It expresses that no two elements in the list have equal *data*-fields. The definition comprises both directions, i.e. both elements reachable from v and elements from which v is reachable. This actually makes it easier to reestablish the property when manipulating the list. It is a formalization of the second data structure invariant for lists. *validSet* does not help to increase precision. It only increases the readability of the output structures.

To capture our notion of set membership we define the *isElement*-predicate. v_1 is an element of set v_2 if its *data*-field is equal to one of the nodes reachable from v_2. Our analysis shows that the effect of the insertion and removal methods on set membership, expressed by *isElement* conforms to the ADT Set axioms.

Our input structures cover all possible lists representing sets pointed to by *set*. *element* points to the element that shall be inserted into the set. Figure 5 displays these structures. In (a) *set* is empty. In (b) *set* is non-empty and set membership of *element* is unknown, *isElement*'s value is indefinite for the nodes pointed to by *element* and *set*.

Insertion Running the analysis for insertion yields three output structures that are shown in Figure 6. All of the resulting structures fulfill the data structure invariants, i.e. $noeq[deq, n]$ is true for the set and $c[n]$ is false everywhere. Also, *isElement* is true for the nodes pointed to by *element* and *set*. In addition, the $or[n, set]$-predicate indicates that elements which were formerly reachable from *set* are still reachable after the execution of *setInsert*.

Looking at the structures one can identify the different cases that the insertion method has to deal with. Structure (a) corresponds to the empty set as input structure. In structure (b) a new element had to be appended to the list, because the *data*-field of *element* is not equal to any of the original elements of the list (the *deq* predicate is false). In structure (c) *element* was already contained in the list, indicated by the *isElement*-predicate.

Removal When translating the C code into a Control Flow Graph in TVLA, we omitted the deallocation of the element in the list. This is only for illustration purposes.

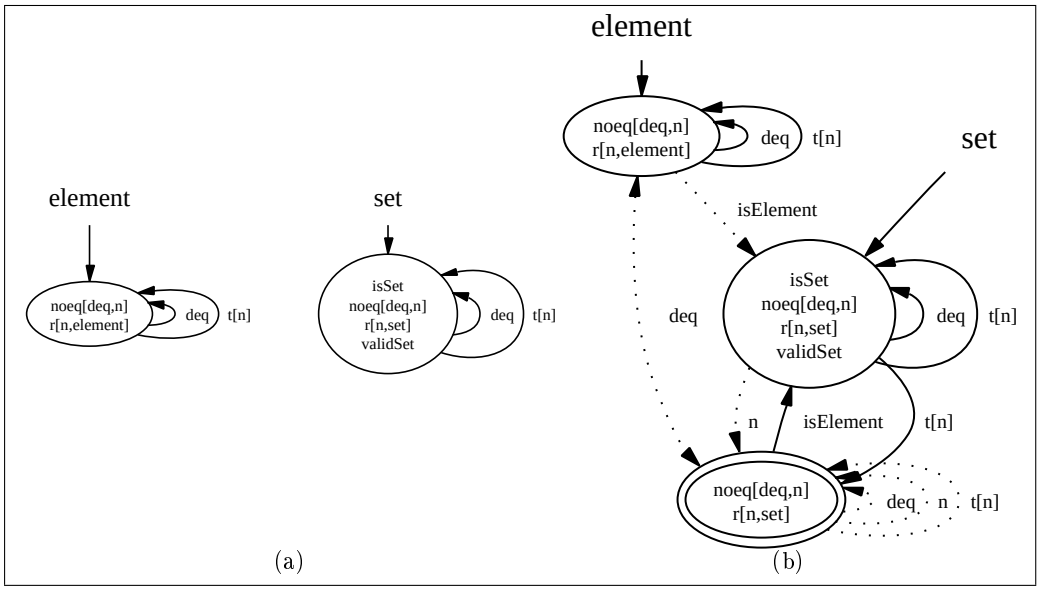


Fig. 5. Input Structures for List-based Insertion and Removal

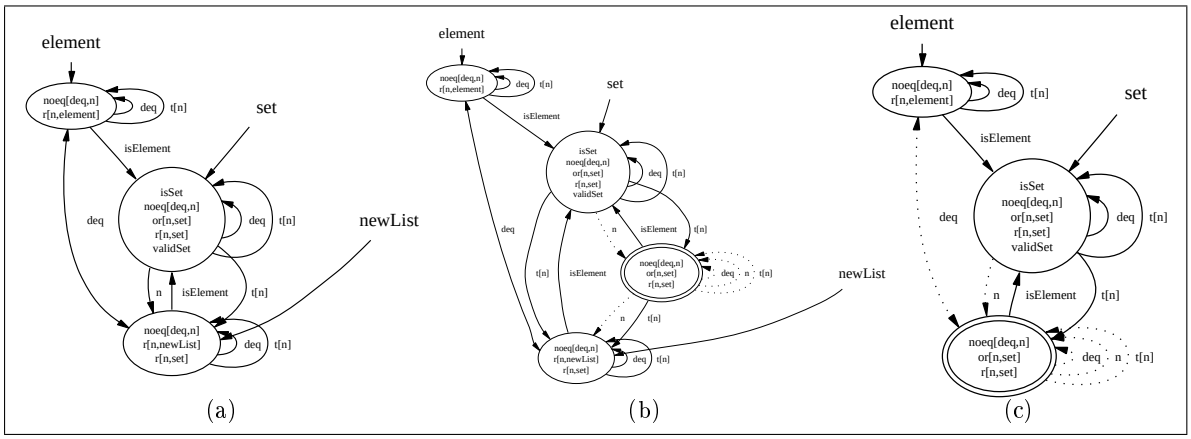


Fig. 6. Output Structures for List-based Insertion

Running *setRemove* results in four output structures displayed in Figure 7. Again, the maintenance of the data structure invariants is proven: $noeq[deq, n]$ is true and $c[n]$ is false everywhere. The element has indeed been removed from the list. This can be observed by the *isElement*-predicate. Other elements of the set are still contained, as indicated by the $or[n, set]$-predicate.

Structures (a) and (c) correspond to the case where *element* was not contained in the set before. The two other structures (a) and (d) reflect the case where *element* was indeed part of the set. The abstraction also distinguishes between empty (c and d) and non-empty sets (a and b).

Membership Test We omit to display the output structures of this analysis, since the routine is not manipulating the heap at all. The analysis checked that our *isElement* function returns true if and only if the *isElement*-predicate holds. This is done by separating the structures into those that reach a point where true is returned and those structures that reach a point where false is returned. By this, we establish a connection between the different analyses. The two other analyses

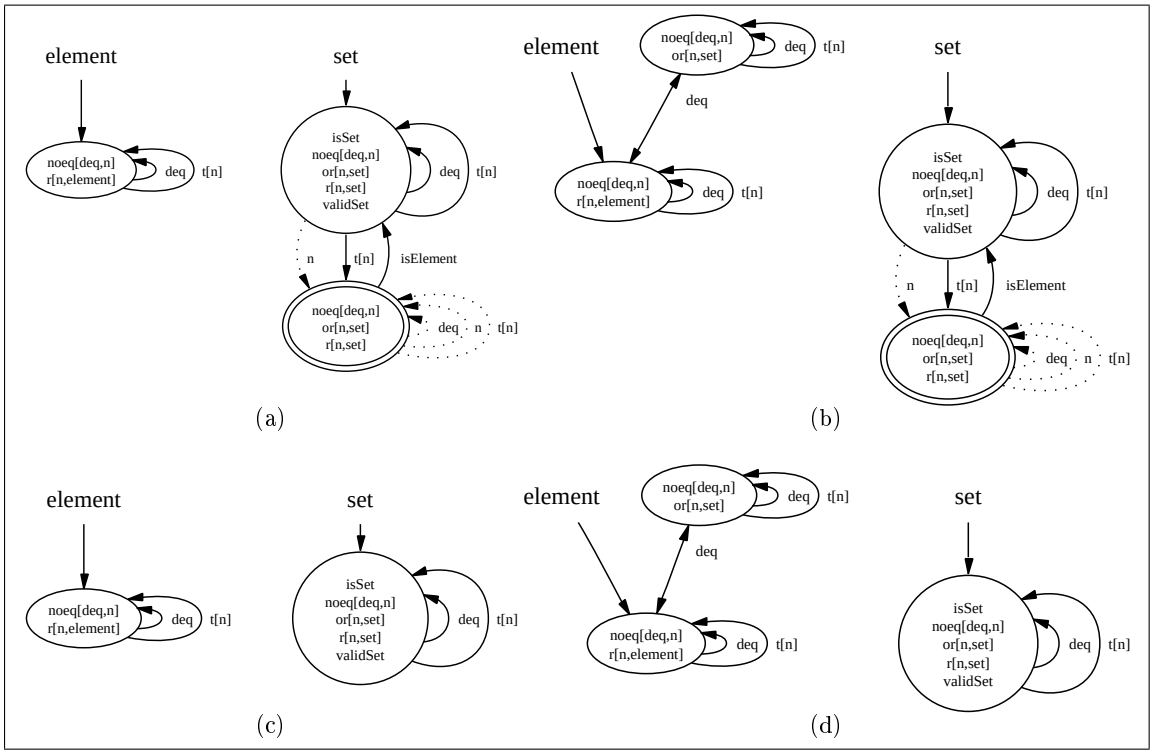


Fig. 7. Output Structures for List-based Removal

on list insertion and removal only proved correctness in terms of the *isElement*-predicate. The current analysis shows that this was just.

Shape Analysis of Tree-based Implementation The domain is represented in a similar way as in the list-based case. Instead of having a *next*-predicate, $left$- and *right*-predicates are used to model the $left$- and *right*-fields in the tree. The $left$-predicate is also used to model the *tree*-field in the set structure to minimize the number of predicates. The *tree*-field only occurs at most once in all of the structures.

Predicate	**Intended Meaning**
$x(v)$ for each $x \in Var$	Pointer variable x points to heap cell v.
$sel(v_1, v_2)$ for each $sel \in \{left, right\}$	The $left$ ($right$) selector of v_1 points to v_2.
$dle(v_1, v_2)$	v_1->$data \leq v_2$->$data$.
$or[x](v)$ for each $x \in Var$	v was reachable from x via $left$- and $right$-fields.
$isSet(v)$	v represents a set.

As noted in the implementation section, an ordering relation is needed here. It is modeled by the *dle*-predicate, which is assumed to be reflexive and transitive during the analysis. $or[x]$ and $isSet$ have the same meaning as before.

While the core predicates used to model the domain were very similar to the list-based case, the choice of instrumentation predicates was quite different. We separate them into two parts. One is solely concerned with the structure of the trees. The other also deals with ordering.

Predicate	Defining Formula	Intended Meaning
$down(v_1, v_2)$	$left(v_1, v_2) \vee right(v_1, v_2)$	The union of the two selector predicates $left$ and $right$.
$downStar(v_1, v_2)$	$down^*(v_1, v_2)$	Records reachability between tree nodes.
$downStar[sel](v_1, v_2)$ for each $sel \in \{left, right\}$	$\exists v.(sel(v_1, v) \wedge down^*(v, v_2))$	Remembers the first selector needed to reach v_2 from v_1.
$r[x](v)$ for each $x \in Var$	$\exists v_1.(x(v_1) \wedge downStar(v_1, v))$	v is transitively reachable from x.
$treeNess$	$\forall v_1, v_2, v.((downStar[left](v, v_1) \wedge downStar[right](v, v_2)) \Rightarrow (\neg downStar(v_1, v_2) \wedge \neg downStar(v_2, v_1)))$	The heap consists of trees.

The two $downStar[sel]$-predicates record reachability between tree-nodes, where the first selector on the path is sel. In ordered trees this determines the relation between the elements in the tree. To be able to check whether the ordering is maintained, it is important to keep this relation precise for elements that are manipulated. $treeNess$ records the first data structure invariant mentioned in the implementation section. We decided to make $treeNess$ a global nullary predicate to reduce the size of the domain. There is a drawback to this approach however. It is nearly impossible to reestablish the property once it is violated, because we lose information about parts of the heap that still satisfy the property. A unary $treeNess$ predicate would be able to capture local violations and make it easier to reestablish the property after it was temporarily destroyed. The methods that we checked maintain $treeNess$ in the entire heap permanently allowing to use the nullary predicate.

Predicate	Defining Formula	Intended Meaning
$dle[x, left](v)$ for each $x \in Var$	$\exists v_1.(x(v_1) \wedge dle(v, v_1) \wedge \neg dle(v_1, v))$	The $data$-field of v is less than the $data$-field of v_1, where v_1 is pointed to by x.
$dle[x, right](v)$ for each $x \in Var$	$\exists v_1.(x(v_1) \wedge \neg dle(v, v_1) \wedge dle(v_1, v))$	The $data$-field of v is greater than the $data$-field of v_1, where v_1 is pointed to by x.
$inOrder[dle]$	$\forall v_2, v_4.(downStar[left](v_2, v_4) \Rightarrow (dle(v_4, v_2) \wedge \neg dle(v_2, v_4))) \wedge \forall v_2, v_4.(downStar[right](v_2, v_4) \Rightarrow (\neg dle(v_4, v_2) \wedge dle(v_2, v_4)))$	All the trees in the heap are in order.
$isElement(v_1, v_2)$	$isSet(v_2) \wedge \exists v_{equal}.(downStar(v_2, v_{equal}) \wedge dle(v_{equal}, v_1) \wedge dle(v_1, v_{equal}) \wedge v_{equal} \neq v_2$	v_1 is an element of set v_2.

The $dle[x, sel]$ captures the relation between the node pointed to by x and other heap nodes. These predicates are used to partition the heap into elements less than the node pointed to by x and those that are greater. Being unary predicates they can be used as abstraction predicates. This could be called a "pseudo-binary abstraction", since parts of the binary predicate dle are taken to form several unary predicates.

$inOrder[dle]$ formalizes the second data structure invariant for ordered trees. It requires elements in the left subtree of a node to be smaller and elements in the right subtree to be greater than the node itself. Smaller and greater are expressed in terms of dle.

The set membership property $isElement$ is formalized similarly to the list-based case. v_1 is an element of set v_2 if its $data$-field is equal to one of the nodes reachable from v_2, where equal can

be formulated in terms of *dle*.

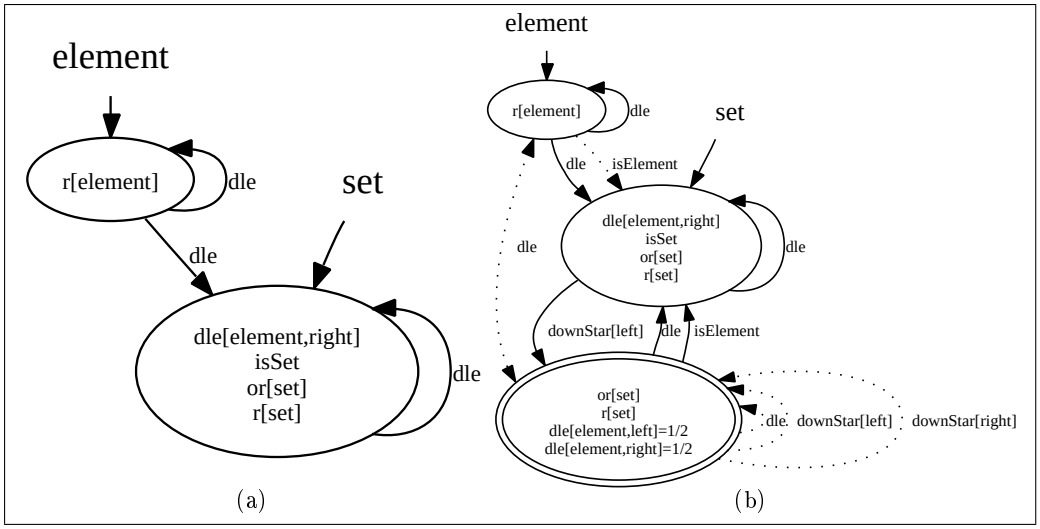


Fig. 8. Input Structures for Tree-based Insertion and Removal

Figure 8 displays the input structures for our analysis of the insertion and removal methods. In the following we omitted several predicates to make the visualizations more readable. The predicates that we left our were $left, right, down, downStar$. Again, we want to cover all possible sets by these abstract structures. In structure (a) *set* is empty and thus *element* is not an element of *set*. Structure (b) represents non-empty sets. *element* might be part of the set, indicated by the dotted *isElement*-predicate and the dotted *dle*-predicate between *element* and the contents of *set*. We also had to assign a value to the *dle*-predicate for *set* which does not have a *data*-field. Its *data*-field is assumed to be greater than all other *data*-fields. Elements that were originally reachable from *set* are marked with $or[set]$ as in the list-based case.

Insertion Running the analysis for set insertion yields 21 structures at exit. Most of them concern special cases where the element had to be inserted in the left- or right-most position of the tree or where the left or the right subtree of the root was empty. All resulting structures fulfilled the data structure invariants and *element* had been inserted into *set*. We picked two structures that represent the most general cases. They can be seen in Figure 9.

Due to the number of binary predicates involved in the analysis the output structures are hard to read. Also, the visualization engine does not know our intuition behind the different predicates, which could help to generate more readable output. In structure (a) the algorithm found a node in the tree that is equal to *element*. The three summary nodes make up the rest of the tree. The summary node to the right represents the subtree of the node that was found. The other two summary nodes partition the parents and neighbors into those that have a smaller *data*-field and those that have a greater *data*-field. For this particular case the partitioning of the set is not important. For structure (b) however it is the key to proving that the ordering is preserved. Here, no node in the tree was found that was equal to *element*. Therefore a new heap node was allocated and inserted into the tree, preserving the ordering. This is were the partition into smaller and larger elements becomes important. Nodes that are greater than the new node can only reach it via a path that starts by going left: $downStar[left]$ is indefinite and $downStar[right]$ is false.

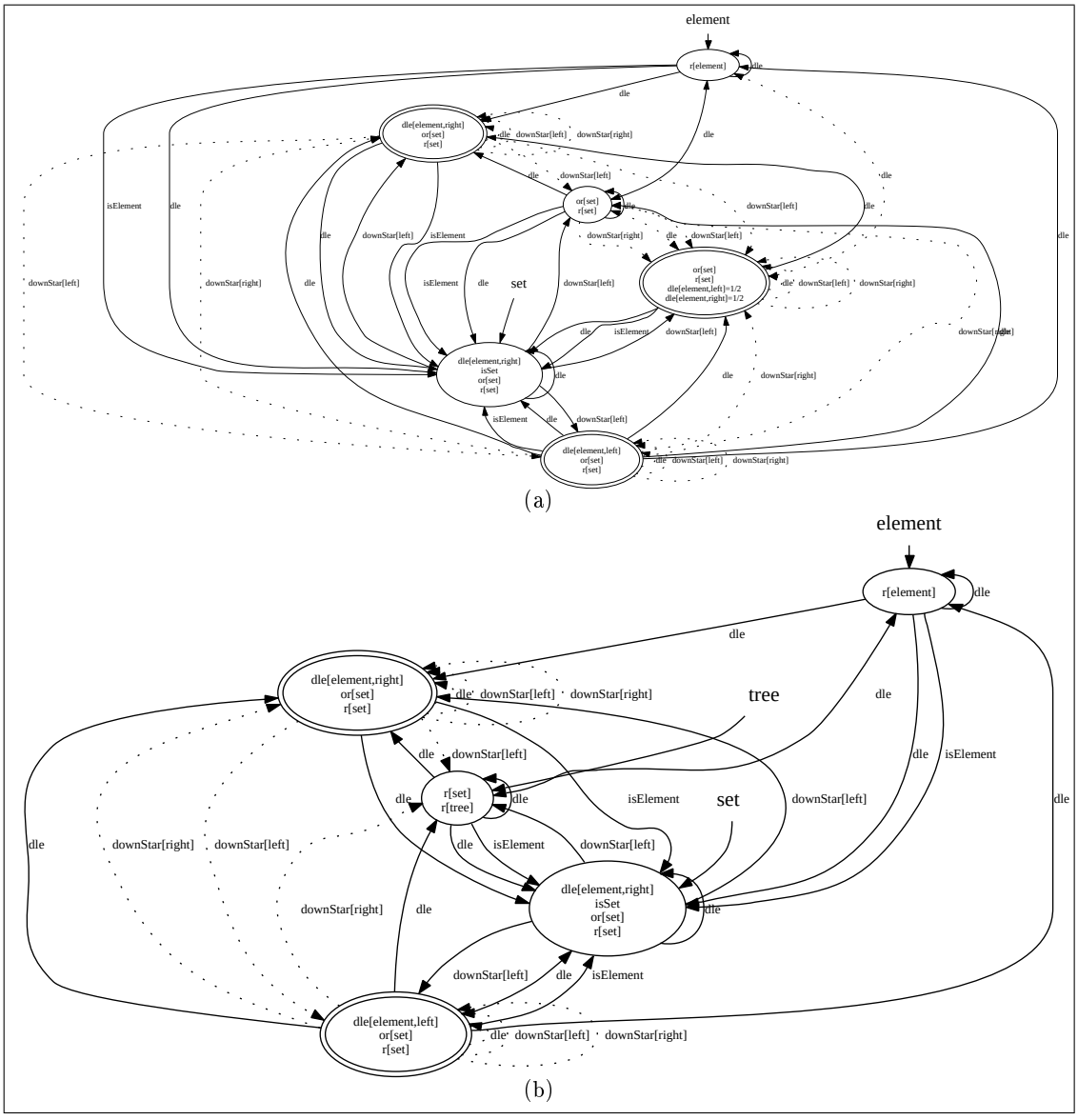


Fig. 9. Sample Output Structures for Tree-based Insertion

Nodes with a smaller *data*-field can in turn only reach it via a path that starts with a *right*-edge ($downStar[right] = 1/2$ and $downStar[left] = 0$).

Removal As noticed in the implementation section, tree-based removal was the most complicated routine that we analyzed. Its size and complexity led to very time-consuming analyses that did not allow a trial and error approach when choosing the abstraction predicates. We used the same predicates as in the analysis of the insertion algorithm. They were developed for this method though and proved to work for the simpler insertion routine, too.

Proving that *element* is not a member of *set* after the analysis was simple, once the data structure invariants could be established. The ordering property ensures that every element only occurs once in the tree. Showing that the ordering data structure invariant was maintained was more difficult. The key predicates involved in proving this were $dle[x, sel]$ and $downStar[sel]$. The use of these predicates in the insertion routine already hints at why they are useful for removal. Figure 4 illustrates the different possibilities when removing an element from the tree. As the algorithm keeps track of the relevant nodes (those represented by circles in the figure) in the graph through pointer variables, $dle[x, sel]$ delivers the necessary partition to keep relevant ordering information. In addition $downStar[sel]$ captures the important first selectors on paths between these parts of the tree.

To cope with the long analysis times we decomposed the problem into smaller ones first:

- Finding the element to delete.
- The element has one or no children.
- The element has two children, the most difficult case.

In the end we put everything together.

Again, we decided to present only two representative output structures out of overall eight. They are shown in Figure 10. Both structures satisfy the two data structure invariants modeled by $inOrder[dle]$ and $treeNess$. In structure (a) *element* was contained in *set* and therefore removed from it. For demonstration purposes we did not free the element taken from the tree. One can see that the tree has been partitioned into nodes with a greater *data*-field and nodes with a smaller *data*-field than *element*. The same holds for structure (b). In this case *element* was not contained in *set* at the invocation of the routine. No node was removed from the tree.

Membership Test Again, we omit to display the output structures. It is quite obvious that the analysis succeeds, because the tree traversal analyzed is part of the insertion and removal methods as well, which were analyzed before.

Empirical Results Table 2 presents some data about the four analyses. The analysis of the insertion, removal and membership test methods of our list-based implementation resulted in a similar number of structures and relatively short analysis times. In the tree-based case, however, the difference was considerable. This can probably be explained with the higher number of unary predicates in the removal analysis, which led to more structures per location. The worst-case complexity of the analysis is doubly-exponential in the number of abstraction predicates. Additionally, the control flow graph (see Figure 11) for removal contains more than three times as many locations as the CFG for insertion.

Discussion We managed to show interesting properties of list- and tree-based set implementations. Our analyses assumes data structure invariants specific to the respective implementation to hold at the entrance. The maintenance of these invariants throughout the execution of the routines

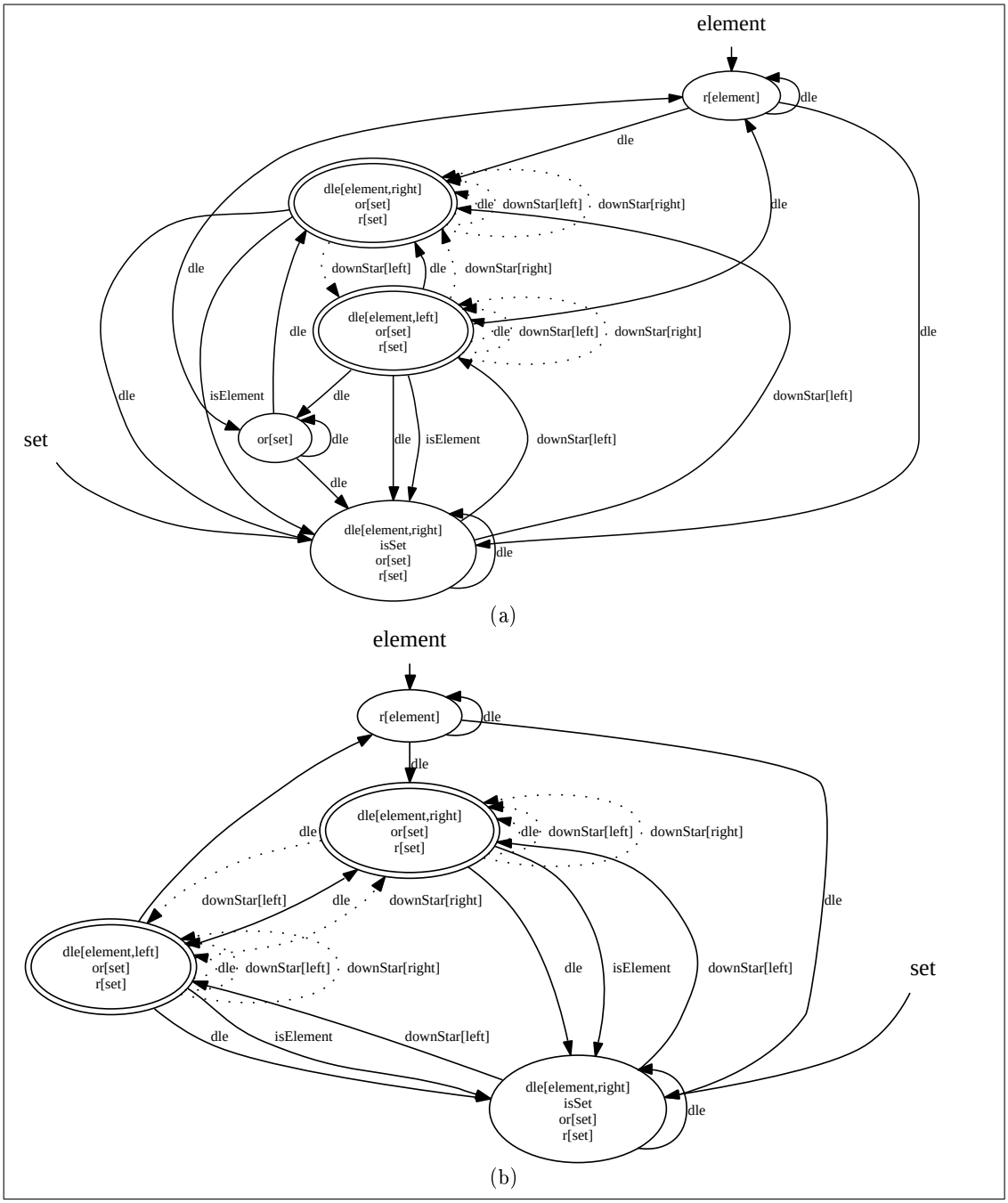


Fig. 10. Sample Output Structures for Tree-based Removal

Analysis	**#locations in CFG**	**#unary predicates**	**#binary predicates**	**#structures**	**average #structs per location**	**maximal #structs per location**	**time**
Membership, List-based	9	20	5	28	3	6	2.570s
Insertion, List-based	19	29	5	81	4	11	2.720s
Removal, List-based	22	29	5	124	5	11	4.050s
Membership, Tree-based	10	18	11	84	8	19	32.84s
Insertion, Tree-based	25	24	11	536	21	91	69.23s
Removal, Tree-based	76	42	11	27697	364	3132	21767s

Table 2. Empirical Results

is established. Using these invariants our analysis was able to prove that the effect of the insertion and removal methods complies with axioms of the ADT Set. The nature of the shape analysis framework limited our proofs to partial correctness.

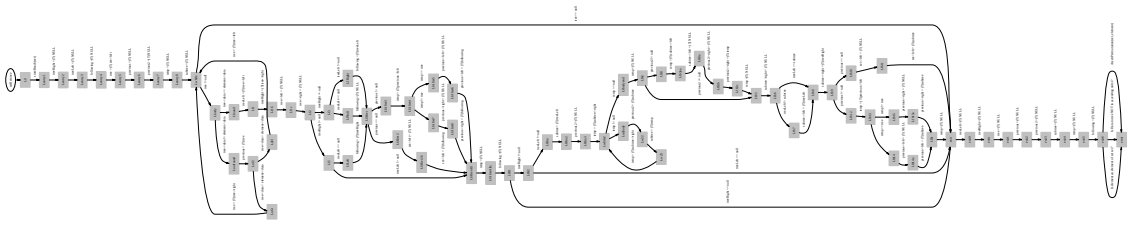


Fig. 11. CFG for Tree Removal

We used the *isElement*-predicate to relate different analyses. While the insertion and removal methods were proved correct in terms of *isElement*, the analysis of the set membership routine showed the equivalence of this routine with *isElement*. This approach loosely corresponds to the abstraction mechanism used in [LKR04]. They use sets to abstract from more complex data structures, which limits them to statically allocated data structures. Our use of *isElement* on the other hand allows to handle dynamically allocated sets.

Choosing the right instrumentation predicates required a thorough understanding of the data structures involved. For trees this meant identifying that reachability alone is not very interesting, but that the first edge on a path from one node to another is important. However, the predicates are not tailored to specific algorithms, but to the underlying data structures. They might prove useful for other algorithms on trees and lists as well.

Abstraction Expressions The need to partition the trees into smaller and larger elements led to the introduction of the $dle[x, sel]$-predicate family. The effect of these unary predicates on the abstraction could also be achieved by using the binary *dle*-predicate in the abstraction process.

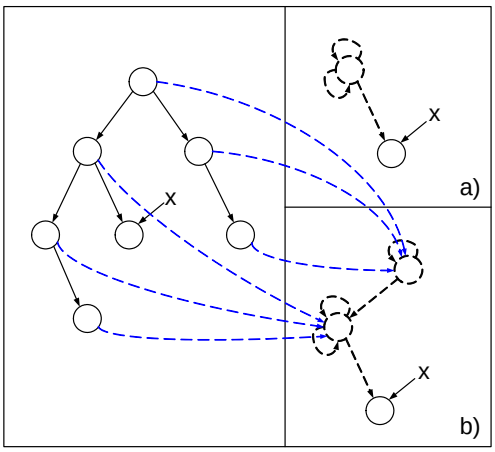


Fig. 12. Abstraction Expressions

Here, individuals should only be joined if they have the same canonical name and if they agree on binary abstraction predicates to other canonical names. This is illustrated in Figure 12. The tree on the left is supposed to be in order. The ordering predicate is not visualized to make it more readable. Canonical Abstraction would collapse all the nodes not pointed to by x (a). The relation between the resulting summary node and the node pointed to by x would be indefinite. Additionally abstracting from *dle* would instead create two summary nodes and keep ordering information definite. Of course, the proposed abstraction can also be achieved using a number of unary abstraction predicates. The number of predicates needed for this is linear in the number of abstraction predicates though, to cover all canonical names.

We propose to specify the abstraction through *Abstraction Expressions*:

Definition 1 (Syntax of Abstraction Expressions). *The set of* Abstraction Expressions *over a set of unary predicates U and a set of binary predicates B is defined inductively as follows:*

- $\{u_1, \ldots, u_n\}$ *is an abstraction expression if* $\{u_1, \ldots, u_n\} \subseteq U$,
- $AE_1 \wedge AE_2$ *is an abstraction expression if* AE_1 *and* AE_2 *are abstraction expressions,*
- $AE \triangleright \{b_1, \ldots, b_n\}$ *is an abstraction expression if* AE *is an abstraction expression and* $\{b_1, \ldots, b_n\} \subseteq B$.

We define the semantics of *Abstraction Expressions* by giving an associated equivalence relation. The equivalence relation determines which nodes are to be merged.

Definition 2 (Semantics of Abstraction Expressions). *The associated equivalence relation* $\sim_{AE}$ *to an* Abstraction Expression AE *is defined inductively as follows:*

- $x \sim_{\{u_1,\ldots,u_n\}} y :\Leftrightarrow \bigwedge_{u \in \{u_1,\ldots,u_n\}} u(x) = u(y)$,
- $x \sim_{AE_1 \wedge AE_2} y :\Leftrightarrow x \sim_{AE_1} y \wedge x \sim_{AE_2} y$,
- $x \sim_{AE \triangleright \{b_1,\ldots,b_n\}} y :\Leftrightarrow x \sim_{AE} y \wedge \bigwedge_{b \in \{b_1,\ldots,b_n\}} \forall z.(\bigsqcup_{\{w \mid w \sim_{AE} z\}} b(x,w) = \bigsqcup_{\{w \mid w \sim_{AE} z\}} b(y,w))$.

The *Abstraction Expression* $\{u_1, \ldots, u_n\}$ is equivalent to *Canonical Abstraction* over $\{u_1, \ldots, u_n\}$. The abstraction depicted in case (b) of Figure 12 can be specified using the *Abstraction Expression* $\{x\} \triangleright \{dle\}$. It will be interesting to see whether there are more applications, where abstraction can be specified more easily using such expressions than by plain *Canonical Abstraction*.

Dead Predicates To speed up the analyses we included additional actions in the control flow graphs of the tree-based programs. These actions nullified certain variables and allowed the engine to collapse structures that were otherwise isomorphic. This was only done for unary predicates representing dead variables, i.e. predicates that further steps of the analysis did not rely on. These predicates could be called dead predicates. A similar effect could have been achieved by marking these predicates as non-abstraction predicates locally. This approach was previously described in Roman Manevich's Master Thesis [Man03]. These dead predicates could be determined by a preceding static analysis. At the time the analyses were conducted it had not been integrated into TVLA yet. We believe that it may dramatically increase the performance of analyses in larger programs that contain many loosely coupled sections. Unfortunately, we cannot give experimental results about the magnitude of the effect. Our analysis for the tree-based removal method did not terminate within days without this optimization. Of course, the optimization could also decrease precision, because more structures are collapsed, possibly losing relevant information. However, in such a case it seems that the wrong abstraction is used, but the analysis succeeds by coincidence.

4 Conclusion

We created a precise shape analysis for programs that are manipulating ordered trees. It is particularly tailored to invariants of the tree data structure. Choosing the right instrumentation predicates required a thorough understanding of the data structures involved. This meant identifying that reachability alone is not very interesting, but that the first edge on a path from one node to another is important. We implemented the analysis in TVLA [LA00,LAS00] and successfully applied it to methods of the tree-based set implementation. The analysis proved that the implementation complies to the axioms (3) and (4) of the ADT Set specification.

$$a \in s.\texttt{insert}(b) \leftrightarrow a =_{el} b \vee a \in s, \quad (3)$$
$$a \in s.\texttt{remove}(b) \leftrightarrow a \neq_{el} b \wedge a \in s \quad (4)$$

We used the *isElement*-predicate to relate different analyses. Our analyses of the insertion and removal methods established the two axioms in terms of *isElement*. Another analysis then established the equivalence between *isElement* and the set membership method $\cdot.\texttt{insert}(\cdot)$. Adapting existing analyses for singly-linked lists allowed us to show the same property for our list-based set implementation.

Inspired by a family of instrumentation predicates used in our tree analysis, we propose a new way of specifying abstractions by so-called "Abstraction Expressions". These expressions allow to not only use unary but also binary predicates in the abstraction specification. "Abstraction Expressions" have the same expressive power as *Canonical Abstraction*. However, we need a smaller number of predicates to express certain abstractions.

5 Future Work

We successfully analyzed a tree-based set implementation. Since the analysis is tailored to the underlying data structure and not to the specific algorithms employed, it might be possible to analyze other algorithms working on trees using the same abstraction.

The tree structure lends itself naturally to recursion. We could possibly combine recent work on interprocedural shape analysis [RS01] with our abstractions to be able to analyze recursive implementations. Modern data structure libraries usually contain more efficient set implementations using balanced trees, like AVL or red-black trees. They maintain even more complicated data structure invariants than the unbalanced tree implementation we analyzed. Algorithms on these structures can usually be implemented more easily using recursion, too. Extending our analysis to

cope with the invariants of balanced trees might make such algorithms amenable as well.

Abstraction Expressions seem useful where we want to distinguish individuals if they differ by binary predicates originating from individuals that we distinguish. In our tree-based analysis, we could separate smaller and larger tree elements. In the shape analysis for RESET, we could use the set membership relation to separate individuals in terms of the sets they belong to. An implementation of the concept would allow deeper insight into the usefulness of the approach.

References

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 296–310, New York, NY, USA, 1990. ACM Press.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification I.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[EM90] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, New York, NY, USA, 1996. ACM Press.

[LA00] Tal Lev-Ami. TVLA: A framework for kleene based static analysis. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2000.

[LARSW00] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 26–38, New York, NY, USA, 2000. ACM Press.

[LAS00] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 280–301, London, UK, 2000. Springer-Verlag.

[LEW97] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of abstract data types.* John Wiley & Sons, Inc., New York, NY, USA, 1997.

[LKR04] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking using set interfaces and pluggable analyses, 2004.

[Man03] Roman Manevich. Data structures and algorithms for efficient shape analysis. Master's thesis, Tel-Aviv University, School of Computer Science, Tel-Aviv, Israel, January 2003. Available at www.cs.tau.ac.il/rumster/msc_thesis.pdf.

[Mic04] Sun Microsystems. Java 2 platform standard edition 5.0 api specification, 2004. Available at http://java.sun.com/j2se/1.5.0/docs/api/.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA - A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, 1999.

[MS96] David R. Musser and Atul Saini. *STL tutorial and reference guide*, volume - of *Addison-Wesley professional computing ser.* Addison-Wesley, 1996.

[Rei05] Jan Reineke. Shape analysis of sets. Master's thesis, Universität des Saarlandes, Germany, June 2005. Available at http://rw4.cs.uni-sb.de/ reineke/publications/MasterReineke.pdf.

[RS01] Noam Rinetzky and Mooly Sagiv. Interprocedural shape analysis for recursive programs. *Lecture Notes in Computer Science*, 2027:133–149, 2001.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3–valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3–valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

Using Abstraction in Modular Verification of Synchronous Adaptive Systems

Ina Schaefer and Arnd Poetzsch-Heffter

{inschaef|poetzsch}@informatik.uni-kl.de
Software Technology Group
Technische Universtiät Kaiserslautern
Germany

Abstract. Self-adaptive embedded systems autonomously adapt to changing environment conditions to improve their functionality and to increase their dependability by downgrading functionality in case of failures. However, adaptation behaviour of embedded systems significantly complicates system design and poses new challenges for guaranteeing system correctness, in particular vital in the automotive domain. Formal verification as applied in safety-critical applications must therefore be able to address not only temporal and functional properties, but also dynamic adaptation according to external and internal stimuli.
In this paper, we introduce a formal semantic-based framework to model, specify and verify the functional and the adaptation behaviour of synchronous adaptive systems. The modelling separates functional and adaptive behaviour to reduce the design complexity and to enable modular reasoning about both aspects independently as well as in combination. By an example, we show how to use this framework in order to verify properties of synchronous adaptive systems. Modular reasoning in combination with abstraction mechanisms makes automatic model checking efficiently applicable.

1 Introduction

In the automotive sector, self-adaptive embedded systems are used for instance as antilock braking (ABS), vehicle stability control (VSC), and adaptive cruise control (ACC) systems. They autonomously adapt to changing environment conditions in order to meet high quality requirements, e.g. to offer the best possible service in any kind of driving condition. Furthermore, adaptation increases dependability and fault-tolerance of systems by autonomously up- and downgrading the functionality according to the available resources. This can for instance be changing qualities of environment sensors. However, adaptation in embedded systems significantly complicates system design and poses new challenges for guaranteeing system correctness, in particular vital in the automotive domain. Therefore, formal verification as applied in safety-critical applications must be able to address not only temporal and functional properties, but also dynamic adaptation according to external and internal stimuli.

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/699

In this paper, we introduce a formal semantic-based framework to model, specify and verify functional and adaptation behaviour of self-adaptive embedded systems. The modelling framework is based on state-transition systems and describes adaptation of module behaviour in terms of an adaptation aspect on top of a set of predetermined module configurations. Restricting adaptation to predetermined reconfiguration makes systems predictable and improves analyses results. Our models are synchronous systems as those can capture simultaneously invoked actions by true concurrency. Most approaches formalizing self-adaptation [1] so far focus on structural and architectural adaptation such as adding and removing components instead of behavioural adaptation of single system modules. Furthermore, they intertwine functionality and adaptation. In contrast, the proposed modelling framework decouples functional and adaptive behaviour and provides a clear formal account of both aspects in separation. This reduces the design complexity and enables explicit and uniform reasoning about purely functional, purely adaptive as well as combined properties. We develop a high level modelling framework in which the special features of dynamic reconfiguration, i.e. the behavioural adaptation and the separation of adaptation and functionality, can be observed and reasoned about directly. If these special properties of the considered class of systems would be encoded into another formalism, this high level specific properties are typically lost and cannot be exploited for tailored analyses.

On top of the formal model, we define a specification logic that allows to express functional, adaptive and temporal properties of the system. Since we can describe the behaviour of our systems by a set of execution traces we will adopt a variant of first-order LTL [3] for our purposes. The proposed framework enables modular reasoning through modular specification of systems. A global system property can be decomposed into local properties of single modules entailing the global property. Furthermore, the model allows to incorporate abstraction mechanisms, for instance to reduce unbounded data domains to finite discrete domains. Modularity combined with appropriate abstraction mechanisms facilitates the efficient integration of existing automatic verification techniques such as model checking into the verification process of synchronous adaptive systems. Thus, the verification effort can be reduced by discharging sub-proof goals automatically.

In this paper, we present the application of our modelling, specification and verification framework at an example system confronted with changing qualities of sensor values. This scenario is quite common in the context of embedded automotive systems. Due to restricted hardware resources, the system has to deal with changing sensor qualities by adapting its functionality to the available resources instead of halting the system. Redundant hardware is not applicable due to the inherent limitations in embedded systems. We show how to model such a sensor quality adaptation as synchronous adaptive system. Afterwards, we verify the safety property that despite problems with the sensors the quality of the system output is below a threshold only for a restricted period of time. This exemplary verification shows the use of modular reasoning and abstraction

techniques and gives an intuition which mechanisms are necessary for efficient automatic verification of synchronous adaptive systems.

The paper is structured as follows: Section 2 gives a short overview of related work on formal analysis of self-adaptive systems. In Section 3, we will introduce our formal semantic-based model of synchronous adaptive systems illustrated with the running example. In Section 4, we introduce an LTL based logic for specifying properties over these models. In Section 5, we show how to use abstraction techniques and modular verification in order to proof the safety property over the running example, before we conclude the paper in Section 6 with an outlook to future work.

2 Related Work

From a general point of view dynamic adaptation is a very diverse area of research including real time systems [5], agent systems [8] and component middleware [4], just to name a few representatives. There are a number of approaches for modelling self-managed dynamic software architectures in a more or less formal manner, e.g. using graphs, logic or process algebra; for a survey, consult [1]. However, most of these approaches consider mere modelling of systems instead of their verification. Additionally, the focus lies mainly on architectural adaptation instead of behavioural adaptation as considered here. Moreover, adaptive and functional behaviour are often intertwined which does not allow separated reasoning about both aspects.

In [10], models of adaptive synchronous systems separate adaptation from functionality by endowing data flow with qualities. The configuration behaviour of one module only depends on the quality transmitted with the input and output variables. Considering the qualities, an abstract model of the adaptation behaviour is extracted from the system which is analysed via model checking. However, functional behaviour is completely discarded whereas our approach allows to reason about adaptive, functional and combined behaviour as in systems where adaptation depends on functional data. In [12], the authors use a model driven approach to modularly define adaptive systems coming close to the modularity considered here. Starting from a global model and global requirements of the overall system, single domains of adaptation are identified which are designed to satisfy local requirements entailing the global ones. However, the notion of adaptivity is more coarse-grained than in synchronous adaptive systems due to three fixed types of adaptation.

With respect to verification of adaptive systems, in [11] a linear temporal logic is extended with an 'adapt' operator for specifying requirements on the system before, during and after the adaptation. In [6], the authors use an approach based on a transitional-invariant lattice. Using theorem proving techniques they show that before, during and after the adaptation the program is always in a correct state in terms of satisfying the transitional-invariants. However, both approaches use a more coarse-grained notion of adaptation than predetermined behavioural reconfiguration as considered here.

3 Formal Models of Synchronous Adaptive Systems

Synchronous adaptive systems are composed from a set of modules where each module has a set of predetermined behavioural configurations it can adapt to. The selected configuration depends on the status of the module's environment. It is determined by an adaptation aspect defined on top of the functional behaviour. The modules are connected via links between input and output variables. Data and adaptation flow are decoupled and do not follow the same links. Adaptations in one module may trigger adaptations in other modules by internal adaptation signals via the adaptation links. That may lead to a chain reaction of adaptations through the system. The systems are assumed to be open systems with non-deterministic input provided by an environment. Furthermore, they are modelled synchronously as their simultaneously invoked actions are executed in true concurrency.

3.1 Running Example

Before we start with the formal definitions, we will illustrate the general behaviour of synchronous adaptive systems at an example system dynamically reconfiguring dependant on the quality provided by its input sensors. Figure 1 shows an overview of the system structure.

The system consists of two modules. They receive input from three sensors and control one actuator. The sensors may produce results with varying quality due to changing environment conditions. Hence, the sensor input is associated with a confidence level. This confidence level is an integer value which reflects the sensor's input quality. The higher the confidence level is the higher is the reliability of the value. In our example, a confidence level below 50 models low confidence, between 50 and 100 medium confidence and above 100 high confidence. The confidence level can be determined by enhancing the mere sensor with a functional module. This module for instance records the sensor values over some period of time and monitors its changes. If the sensor value changes by a great amount over a short period of time confidence in this sensor is reduced. Another possibility to calculate the confidence level may be to monitor other system parameters. By performing a plausibility check the sensor module can infer the confidence of the input.

The first two sensor inputs are fed into the first system module which selects one of the sensor inputs according to their confidences. In the considered scenario, sensor 1 produces very good results reflecting the value to be measured very closely. But sensor 1 is also very likely to produce very bad results because of environment changes. This is reflected in the attached confidence level. If the confidence falls below 50, the value is no longer guaranteed to be good enough. Then, the second sensor becomes important. It measures the same input source as the first sensor in general providing lower confidence. Hence, the first sensor is mostly preferred over the second. However, the second sensor is more robust which is reflected by the assumption that the confidence never falls below 50. Thus, if the first sensor produces data with low confidence over some period of

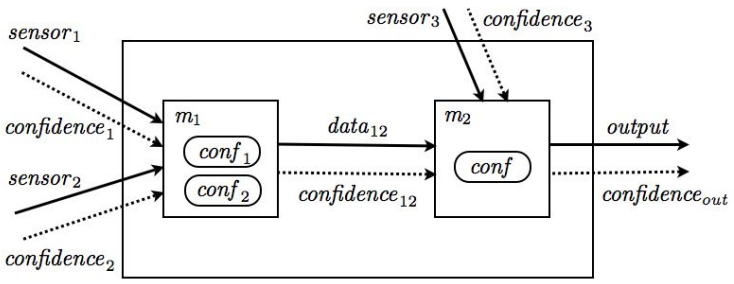


Fig. 1. Graphical Representation of the Running Example System

time the system adapts to use the second sensor in order to ensure sufficient confidence of the output. In detail, the adaptation works as follows: If the confidence level of the first sensor is smaller than 50 for more than 2 subsequent cycles the module switches to the value of the second sensor. Sensor 2 is then used as long as the confidence of the first sensor is smaller than 100. If the confidence of sensor 1 is above 100 for 3 subsequent cycles it is assumed that sensor 1 has recovered. Then, the system will return to using sensor 1 in order to use better quality inputs in general.

According to the selected sensor a different functionality is used to produce the module output. This can for instance be necessary in order to transform the input from a different unit of measurement. The output together with the confidence level of the selected sensor is passed on to the second module which receives another data value and a respective confidence from a third sensor. This sensor is assumed to be of the same type as sensor 2 always producing a medium quality value with confidence above 50. The second module uses its two input values to trigger the actuator. Therefore, it only needs a single configuration. For the confidence level it simply computes the minimum confidence of the received.

An interesting property of this system is that the confidence should never fall below 50 for more than two subsequent cycles. This property depends on the assumption that the second and third sensor are more robust always providing confidence above 50. Ensuring this property is important because the actuator may break down putting the system in a dangerous situation if it gets input with low confidence for more than 2 subsequent cycles. However, it is desirable to use the best possible sensor input. So the adaptation is designed to use sensor 1 whenever appropriate.

3.2 Syntax

In this section, we define the syntax of our formal modelling language for synchronous adaptive systems (SAS). It is based on state-transition systems and

incorporates ideas from aspect-oriented software engineering in order to decouple functional from adaptive behaviour. We assume that we are given a set of variable names *Var* and a set of values *Val*. It would also be possible to enhance this with variable types and associated variable domains. The smallest construction element is a module. It contains a set of different predetermined configurations the module can adapt to dependent on the current status of its environment. The adaptation is realised by an adaptation aspect. Before the execution of the actual functionality the adaptation aspect evaluates the configuration guards and determines the configuration to use.

Definition 1 (Module and Adaptation). *An SAS module m is a tuple $m = (in, out, loc, init, confs, adaptation)$ with*

- *$in \subseteq Var$, the set of input variables, $out \subseteq Var$, the set of output variables, $loc \subseteq Var$, the set of local variables and $init : loc \rightarrow Val$ their initial values*
- *$confs = \{conf_j = (guard_j, next\_state_j, next\_out_j) \mid j = 1, ..., n\}$ the configurations of the module, where*
  - *$guard_j$: a first-order formula over $\{in, loc, adapt\_in, adapt\_loc\}$ determining when the configuration j is applicable*
  - *$next\_state_j$: $(in \cup loc \rightarrow Val) \rightarrow (loc \rightarrow Val)$ the next state function for configuration j*
  - *$next\_out_j$: $(in \cup loc \rightarrow Val) \rightarrow (out \rightarrow Val)$ the output function for configuration j*

The adaptation is defined as a tuple $adaptation = (adapt\_in, adapt\_out, adapt\_loc, adapt\_init, adapt\_next\_state, adapt\_next\_out, adapt\_trigger)$ where

- *$adapt\_in \subseteq Var$, the set of adaptation in-parameters, $adapt\_out \subseteq Var$, the set of adaptation out-parameters, $adapt\_loc \subseteq Var$, the set of adaptation local state variables and $adapt\_init : adapt\_loc \rightarrow Val$ their initial values*
- *$adapt\_next\_state : (adapt\_in \cup adapt\_loc \rightarrow Val) \rightarrow (adapt\_loc \rightarrow Val)$ the adaptation next state function*
- *$adapt\_next\_out_i : (adapt\_in \cup adapt\_loc \rightarrow Val) \rightarrow (adapt\_out \rightarrow Val)$ the adaptation output function*
- *$adapt\_trigger : (in \cup loc \cup adapt\_in \cup adapt\_loc \rightarrow Val) \rightarrow \{1, ..., n\}$ for n the number of configurations*

Because the first module in our running example has the more interesting adaptation behaviour we will focus on this module for illustrating the modelling framework. Module m_1 possesses two functional inputs $sensor_1$ and $sensor_2$ and the functional output $data_{12}$. Furthermore, it receives the confidence levels from sensor 1 $confidence_1$ and from sensor 2 $confidence_2$ as adaptation inputs and produces $confidence_{12}$ as adaptation output propagating the confidence of the selected sensor. A functional local state does not exist because the module solely transforms input to output according to two configurations, namely configuration $conf_1$ standing for the use of sensor 1 and $conf_2$ for the use of sensor 2.

In Figure 2, the adaptation behaviour, as defined by the $adapt\_next\_state_1$ function, is depicted as a state transition diagram. The adaptation local state

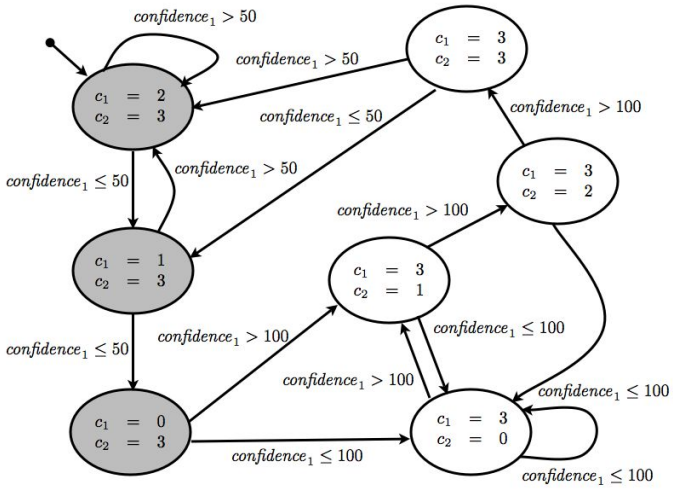


Fig. 2. State Transition Diagram for Adaptation Behaviour of Module 1

consists of two counters c_1 and c_2. If sensor 1 is used counter c_1 counts the subsequent cycles in which $confidence_1$ falls below 50. This counter is initialised to 2. Counter c_2 counts the cycles in which $confidence_1$ is above 100 if sensor 2 is used. It is initialised to 3. The counter c_1 is set to 3 in order to reflect the use of sensor 2. Thus, the guard for use of sensor 1 in $conf_1$ is $c_1 \leq 2$ and the guard for sensor 2 in $conf_2$ is $c_1 = 3$. If sensor 1 is used $confidence_{12} := confidence_1$ and if sensor 2 is used $confidence_{12} := confidence_1$. In Figure 2, the grey circles denote states in which sensor 1 is used and the white ones states where sensor 2 is used.

An SAS system is composed from a set of modules that are interconnected with their input and output variables. The system is an open system with an environment providing non-deterministic input and output via connections from environment input and output to module input and output variables. For technical reasons, we have to assume that the variable names of all modules in a composed system are pairwise disjoint. This can be easily achieved by indexing the module variables with the respective module index. By an injective connection function, we link module output variables to other module's input variables. Furthermore, we link environment input variables to module input variables and module output variables to environment output variables. This means that one variable is connected to one other variable only. If we want to transfer the same output to several places we have to simulate this by duplicating the output variable. Note that in this definition adaptation and functional input and output are decoupled. Adaptation and data flow do not follow the same links such that a module can forward its data to one module and notify a different module to adapt.

Definition 2 (System). *A synchronous adaptive system (SAS) is a tuple*

$$SAS = (M, input_a, input_d, output_a, output_d, conn_a, conn_d)$$

where

- M *is a set of modules* $M = \{m_1, \ldots, m_n\}$ *where* $m_i = (in_i, out_i, loc_i, init_i, confs_i, adaptation_i)$
- $input_a \subseteq Var$ *are adaptation inputs and* $input_d \subseteq Var$ *functional inputs to the system*
- $output_a \subseteq Var$ *are adaptation outputs and* $output_d \subseteq Var$ *functional outputs from the system*
- $conn_a$ *is an injective function connecting adaptation outputs to adaptation inputs and also environment adaptation inputs to module adaptation inputs and module adaptation outputs to environment adaptation outputs, i.e.* $conn_a : (adapt\_out_j \rightarrow adapt\_in_k) \cup (input_a \rightarrow adapt\_in_k) \cup (adapt\_out_k \rightarrow output_a)$ *for* $j, k = 1, ..., n$
- $conn_d$ *is an injective function connecting outputs of modules to inputs and also environment inputs to module inputs and module outputs to environment outputs, i.e.* $conn_d : (out_j \rightarrow in_k) \cup (input_a \rightarrow in_k) \cup (out_k \rightarrow output_a)$ *for* $j, k = 1, ..., n$

We can model the running example as

$$SAS = (M, input_a, input_d, output_a, output_d, conn_a, conn_d)$$

where $M = \{m_1, m_2\}$. The adaptation inputs are $input_a = \{confidence_1, confidence_2\}$ and the functional inputs are $input_d = \{sensor_1, sensor_2\}$. The adaptation outputs are $output_a = \{confidence_{out}\}$ and the functional outputs are $output_d = \{output\}$. The connections between the modules are as depicted in Figure 1.

3.3 Semantics

The semantics of an SAS is defined in a two layered approach. Firstly, we define the local semantics of single modules similar to standard state-transition systems. From this, we secondly give the global semantics of the composed system.

A local state of a module is defined by the evaluation of the module's variables, i.e. the input, output and local variables and the adaptation counterparts. A local state is initial, if its functional and adaptation variables are set to their initial values and input and output are undefined. A local transition between two local states evolves in two stages: Firstly, the adaptation aspect computes its new local state and its new adaptation output from the current adaptation input and the previous adaptation state. The adaptation aspect further selects the configuration with the smallest index and valid guard with respect to the current input and the previous functional and adaptation state. Since the configurations are prioritised according to their index we do not require them to be disjoint. The system designer should ensure that the system has a build-in default configuration which becomes applicable when no other configuration is. The selected configuration is used to compute the new local state and the new output from the current functional input and the previous functional state.

Definition 3 (Local States and Transitions). *A local state s of an SAS module m is defined as evaluation of the module's variables.*

$$s : in \cup out \cup loc \cup adapt\_in \cup adapt\_out \cup adapt\_loc \rightarrow Val$$

A local state s is called initial if $s|_{loc} = init$, $s|_{adapt\_loc} = adapt\_init$ and $s|_V = undef$ for all $V = in \cup out \cup adapt\_in \cup adapt\_out$. A local transition between two local states s and s' is defined as $s \rightarrow_{loc} s'$ iff

$$\begin{gathered} s'|_{adapt\_loc} = adapt\_next\_state(s'|_{adapt\_in} \cup s|_{adapt\_loc}) \\ s'|_{adapt\_out} = adapt\_next\_out(s'|_{adapt\_in} \cup s|_{adapt\_loc}) \\ s'|_{loc} = next\_state_i(s'|_{in} \cup s|_{loc}) \text{ and } s'|_{out} = next\_out_i(s'|_{in} \cup s|_{loc}) \end{gathered}$$

$$\begin{gathered} \text{and} \qquad adapt\_trigger(s'|_{in \cup adapt\_in} \cup s|_{loc \cup adapt\_loc}) = i \\ \text{iff } s'|_{in} \cup s|_{loc} \cup s|_{out} \cup s'|_{adapt\_in} \cup s|_{adapt\_out} \cup s|_{adapt\_loc} \models guard_i \\ \forall\ 0 < j < i, s'|_{in} \cup s|_{loc} \cup s|_{out} \cup s'|_{adapt\_in} \cup s|_{adapt\_out} \cup s|_{adapt\_loc} \not\models guard_j \end{gathered}$$

A global system state is the union of the local states of the system modules together with an evaluation of the system's environment input and output. A global system state is initial if all local states are initial and the system input und output are undefined. A transition between two global states is performed in three stages. Firstly, each module reads its input either from another module's output of the previous cycle or from the environment in the current cycle. Secondly, each module synchronously performs a local transition. Thirdly, the modules directly connected to the system output write their results to the output variables.

Definition 4 (Global States and Transitions). *A global state σ of an SAS consists of the module's local states $\{s_1, \ldots, s_n\}$ where s_i is the local state of $m_i \in M$ and an evaluation of the functional and adaptive input and output, i.e. $\sigma = s_1 \cup \ldots \cup s_n \cup ((input_a \cup input_d \cup output_a \cup output_d) \rightarrow Val)$. A global state σ is called initial if all local states s_i for $i = 1, \ldots, n$ are initial and the system's input and output are undefined. Two states σ^i and σ^{i+1} perform a global transition, i.e. $\sigma^i \rightarrow_{glob} \sigma^{i+1}$ iff*

- *for all $x, y \in Var \setminus (input_d \cup input_a)$ with $conn_d(x, y)$ or $conn_a(x, y)$: $\sigma^{i+1}(y) := \sigma^i(x)$ and for all $x \in input_a$ and $y \in Var$ with $conn_a(x, y)$: $\sigma^{i+1}(y) := \sigma^{i+1}(x)$ and for all $x \in input_d$ and $y \in Var$ with $conn_d(x, y)$: $\sigma^{i+1}(y) := \sigma^{i+1}(x)$*
- *for all $s^i_j \in \sigma^i$ and for all $s^{i+1}_j \in \sigma_{i+1}$ $s^i_j \rightarrow_{loc} s^{i+1}_j$*
- *for all $x \in Var$ and $y \in output_d$ with $conn_d(x, y)$: $\sigma^{i+1}(y) := \sigma^{i+1}(x)$ and for all $x \in Var$ and $y \in output_a$ with $conn_a(x, y)$: $\sigma^{i+1}(y) := \sigma^{i+1}(x)$*

A sequence of global states $\sigma^0\sigma^1\sigma^2 \ldots$ of an SAS is a system trace if firstly σ^0 is an initial global state and secondly, for all $i \geq 0 : \sigma^i \rightarrow_{glob} \sigma^{i+1}$. The set $Runs(SAS) = \{\sigma^0\sigma^1\sigma^2 \ldots \mid \sigma^0\sigma^1\sigma^2 \ldots \text{ is a system trace}\}$ gives the semantics of the SAS.

4 A Logic for Synchronous Adaptive Systems

In this section, we will introduce a specially tailored logic for reasoning about synchronous adaptive systems. The properties of the system behaviour can be classified in three dimensions: functional behaviour, adaptation behaviour and combined properties. Combined properties equally refer to functional and adaptive system aspects, for instance if adaptation depends on functional values. The environment input is assumed to be non-deterministic such that the behaviour of a system can be described by a set of possible execution traces as infinite sequences of states. Hence, we adopt a variant of the linear time logic LTL [3] by adding special basic predicates for the considered systems to standard first-order and LTL connectives.

For a module, we need predicates to describe its local state, the input and output values and the respective adaptation counterparts. Therefore, equality and the less-than-or-equal relation over terms build from the relevant variables are employed. Furthermore, the configuration currently used is described by the predicate $use\_conf_{t_2}(t_1)$ which is true if the module denoted by term t_2 uses the configuration denoted by t_1 in the current state. On system level, we have predicates in order to speak about the connections between output and input variables implemented by the predicates $is\_conn_d(x_1, x_2)$ and $is\_conn_a(x_1, x_2)$ which are true if there is a functional or an adaptive connection between x_1 and x_2.

Definition 5 (Linear $\mathcal{L}_{SAS}$). *The grammar of linear $\mathcal{L}_{SAS}$ is defined as follows:*

$$
\begin{aligned}
t \in Terms &::= x \in Var \mid v \in Val \mid f(t_1, \ldots, t_n) \\
a \in Atoms &::= t_1 = t_2 \mid t_1 \leq t_2 \mid is\_conn_{[a|d]}(x_1, x_2) \mid use\_conf_{t_2}(t_1) \\
\varphi \in StFmlae &::= true \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \exists x.\ \varphi \mid \forall x.\ \varphi \\
\psi \in Fmlae &::= \varphi \mid \mathbf{X}\ \psi \mid \ \mathbf{F}\ \psi \mid \ \mathbf{G}\ \psi \mid \psi_1\ \mathbf{U}\ \psi_2
\end{aligned}
$$

A Formula $\varphi \in \mathcal{L}_{SAS}$ is interpreted over a path π where π is an infinite sequence of global states $\pi = \sigma^0, \sigma^1, \ldots$ which forms a system trace as defined in the previous section. We denote that a system trace of a synchronous adaptive system SAS $\pi \in Runs(SAS)$ is a model for a formula φ by $\pi \models_{SAS} \varphi$.

The interpretation of the temporal and first-order formulae complies to standard first-order LTL semantics [3]. Terms are evaluated by simply extending the variable assignments of a state σ to $\hat{\sigma}$. Equality and less-than-or-equal relation are interpreted standardly. The predicates $is\_conn_a(x_1, x_2)$ and $is\_conn_d(x_1, x_2)$ are valid if there is a connection via the respective connection functions. For reasoning purposes only, we enhance the local state of module i with an additional state variable $config_i$ which captures the configuration that is used in this state. Its value is determined during the global transition $\sigma \rightarrow \sigma'$ by using the *adapt_trigger* function as defined in Definition 1 which selects the applicable configuration.

$$\sigma'(config_i) = adapt\_trigger_i(s'_i|_{in \cup adapt\_in} \cup s_i|_{loc \cup adapt\_loc})$$

Then, we are able to define the predicate *use_conf* over a state σ by $use\_conf_{t_2}(t_1) \equiv true$ iff $\sigma(config_{\hat{\sigma}(t_2)}) = \hat{\sigma}(t_1)$.

Furthermore, the boolean connectives are interpreted standardly. The next operator, $\mathbf{X}\,\varphi$, determines that φ is true in the next state, i.e. over the path π^1 which is the state sequence $\sigma^1, \sigma^2 \ldots$. A formula is globally true, $\mathbf{G}\,\varphi$, iff for all $i \geq 0$, φ holds over π^i, the path π starting in the i-th state. The formula $\mathbf{F}\,\varphi$ is true is there exists $i \geq 0$ such that $\pi^i \models_{SAS} \varphi$. The until operator $\varphi_1\,\mathbf{U}\,\varphi_2$ denotes that there exists $j \geq 0$ such that $\pi^j \models_{SAS} \varphi_2$ and for all $0 \leq i < j$, $\pi^i \models_{SAS} \varphi_1$. We say that a formula $\varphi \in \mathcal{L}_{SAS}$ is valid for an *SAS* if $\pi \models_{SAS} \varphi$ for all paths $\pi \in Runs(SAS)$ and that it is satisfiable if there exists $\pi \in Runs(SAS)$ such that $\pi \models_{SAS} \varphi$.

5 Towards Modular Verification using Abstraction

Having defined a specification logic on top of the formal model we are now able to formally verify properties specified in $\mathcal{L}_{SAS}$. This verification process should incorporate automatic verification techniques such as model checking whenever possible. For an intuition how the proposed framework can be applied we consider the running example of the sensor quality adaptation as described in Section 3.1. The safety property to be shown is that the quality of the output at the actuator is never below 50 for more than 2 subsequent cycles. Otherwise, the actuator may break down causing the system to enter a dangerous situation. In $\mathcal{L}_{SAS}$, this property can be expressed by the formula φ that is required to hold for all paths $\pi \in Runs(SAS)$.

$$\varphi = \mathbf{G}\,\neg(confidence_{out} \leq 50 \wedge \mathbf{X}\,confidence_{out} \leq 50 \wedge \mathbf{X}\,\mathbf{X}\,confidence_{out} \leq 50)$$

For verification we proceed as follows. Firstly, we abstract the unbounded integer domain of the confidence level to three discrete values *low*, *med* (for medium) and *high*. This is necessary because automatic model checking techniques to be applied later can in general only deal with finite state systems. The abstraction has to preserve the properties of the concrete system, i.e. if the abstract property holds over the abstract system, also the concrete property holds over the concrete system. For our example, we can construct an abstract $SAS^{\#}$ along the lines of [2]. We use the following surjection h for mapping concrete confidence integer values to abstract values:

$$h(confidence) = \begin{cases} low \text{ if } confidence \leq 50 \\ med \text{ if } 50 < confidence \leq 100 \\ high \text{ if } confidence > 100 \end{cases}$$

For the paths of the abstract system $SAS^{\#}$ we have to ensure two conditions such that $SAS^{\#}$ approximates *SAS* and preserves its $\mathcal{L}_{SAS}$ properties. Firstly, the set of concrete initial states must be mapped to the set of abstract initials states. Secondly, the concrete transition relation must be contained in the abstract transition relation. In our example, this means that we must abstract all

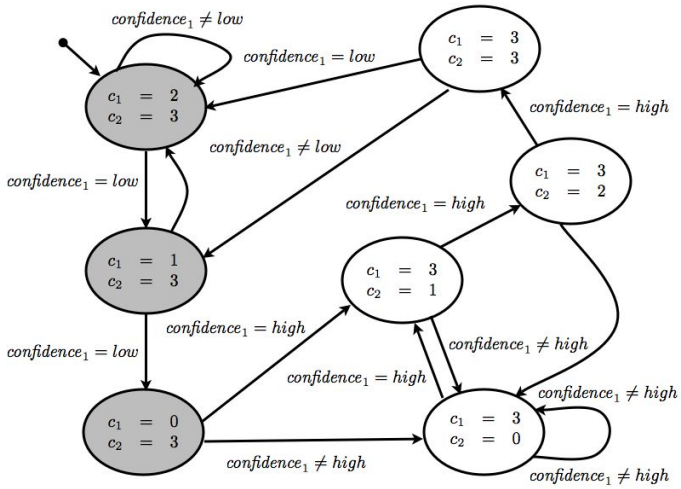


Fig. 3. State Transition Diagram for the Abstract Adaptation Behaviour of Module 1

conditions in configuration guards, adaptation next state and adaptation output functions that depend on $confidence_1$, $confidence_2$, $confidence_{12}$ and $confidence_3$ by the corresponding expressions using the abstract values *low*, *med* and *high*. For an example consider the abstracted transition diagram for module 1 in Figure 3. Additionally, the functions for calculating the confidence outputs have to be abstracted. This is easy for module 1 since it just propagates the relevant confidence already abstract in the abstract system. For module 2, the *adapt_next_out* function is defined as $confidence_{out} := min\{confidence_{12}, confidence_3\}$. Here, we have to give an abstract minimum function $min^{\#}$ which reflects the intuitive ordering that *low* is smaller than *med* which is smaller than *high*. The abstract property reads as follows: $\varphi^{\#} =$

$$\mathbf{G}\,\neg(confidence_{out} = low \wedge \mathbf{X}\,confidence_{out} = low \wedge \mathbf{X}\,\mathbf{X}\,confidence_{out} = low)$$

Verification of the abstract property $\varphi^{\#}$ over the abstract system $SAS^{\#}$ immediately implies validity of φ over SAS by construction of the abstraction using the results of [2].

Secondly, we modularly verify the abstract property over the abstract system. Therefore, we decompose the global property into two local properties over the two modules. From their validity can infer validity of the overall system property. The global property $\varphi^{\#}$ can be decomposed as follows. For module 2, we use the definition of the *adapt_next_out* function and show $\varphi_2^{\#} =$ (where c is used as abbreviation for *confidence*)

$$\mathbf{G}\,\neg(min^{\#}\{c_{12}, c_3\} = low \wedge \mathbf{X}\,min^{\#}\{c_{12}, c_3\} = low \wedge \mathbf{X}\,\mathbf{X}\,min^{\#}\{c_{12}, c_3\} = low)$$

By assumption on sensor 3 that its confidence is always greater than 50 or greater than *low* this property boils down to $\varphi_2^{\#} =$

$$\mathbf{G}\,\neg(confidence_{12} = low \wedge \mathbf{X}\,confidence_{12} = low \wedge \mathbf{X}\,\mathbf{X}\,confidence_{12} = low)$$

This is actually a property over module 1. So it suffices to prove $\varphi_1^{\#} \equiv \varphi_2^{\#}$ over module 1. We can enter this property together with the abstract module description into a model checker, for instance [9]. This will explore all paths of the abstract system and return the result that for all paths π of the abstracted module 1, $\pi \models_{m^{\#}{}_1} \varphi_1^{\#}$. This can also be seen in the abstract transition graph as depicted in Figure 3. If the confidence of sensor 1 is *low* for 2 subsequent cycles the system adapts to use sensor 2. Sensor 2 by assumption has a confidence of greater than 50 or in the abstract greater than low. So, the property $\varphi_1^{\#}$ holds on all execution paths. As a consequence, we can conclude by combining the results of abstraction and modularity that the example *SAS* satisfies the initial property φ.

6 Conclusion and Future Work

In this paper, we have introduced a formal semantic-based framework to model, specify and verify the functional and the adaptation behaviour of synchronous adaptive systems. The modelling framework separates functional and adaptive behaviour in order to reduce the design complexity and to allow modular reasoning about both aspects independently but also in combination. We have shown how to apply this framework for the verification of a safety property by the example of a sensor quality adaptation system.

As we have observed in the running example, modularity combined with abstraction reduces the complexity of sub-proof goals necessary to infer the desired overall system property. For these sub-goals, model checking algorithms such as [9] become efficiently applicable. Hence, for future work, we want to further investigate the use of modular verification in combination with abstraction mechanisms. In this direction, we want to integrate an automatic theorem prover dealing with modularity and abstraction with automatic model checking methods. Furthermore, we plan to equip our modelling framework with means for expressing hierarchy in order to be able to compose complex systems from a number of subsystems and to exploit this hierarchy for verification. In addition to that, we want to implement a translation from UML-like models of synchronous adaptive systems in the GME [7] framework to SAS models in order to provide GME models with a firm semantic basis and to make our approach end-user compatible by a graphical modelling front end.

References

1. J.S. Bradbury, J.R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proc. of the International Workshop on Self-Managed Systems (WOSS'04)*, 2004.
2. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

3. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier, Amsterdam, 1990.
4. W. Gilani, N. Naqvi, and O. Spinczyk. On adaptable middleware product lines. In *Proc. of 3rd Workshop an Adaptive and Reflective Middleware*, page 207213, 2004.
5. O. Gonzalez, H. Shrikumar, J. Stankovic, and K. Ramamritham. Adaptive fault-tolerance and graceful degradation under dynamic hard real-time scheduling. In *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, pages 79–89, 1997.
6. S.S. Kulkarni and K.N. Biyani. Correctness of component-based adaptation. In *Proc. of Intl. Symposium on on Component Based Software Engineering*, pages 48–58, 2004.
7. A. Ledeczi and al. The Generic Modeling Environment. In *Proc. of IEEE International Workshop on Intelligent Signal Processing (WISP)*, 2001.
8. O. Marin, M. Bertier, and P. Sens. DARX - a framework for the fault tolerant support of agent software. In *Proc. of IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 406–418, 2003.
9. K. Schneider and T. Schuele. Averest: Specification, verification, and implementation of reactive systems. In *Proc. of Conference on Application of Concurrency to System Design (ACSD)*, 2005.
10. K. Schneider, T. Schuele, and M. Trapp. Verifying the adaptation behavior of embedded systems. In *Proc. of Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2006.
11. J. Zhang and B.H.C. Cheng. Specifying adaptation semantics. In *Proc. of ICSE 2005 Workshop on Architecting Dependable Systems (WADS 2005)*, pages 1–7, 2005.
12. J. Zhang and B.H.C. Cheng. Model-based development of dynamically adaptive software. In *Proc. of the International Conference on Software Engineering (ICSE'06)*, 2006.