IJCAR 2008

4th International Joint Conference on Automated Reasoning

Sydney, Australia, August 10–15, 2008

Workshop Program

5th International Verification Workshop – VERIFY'08

Bernhard Beckert and Gerwin Klein (Chairs)

WS 1 – August 10/11

Preface

The VERIFY workshop series aims at bringing together people who are interested in the development of safety and security critical systems, in formal methods, in the development of automated theorem proving techniques, and in the development of tool support. Practical experiences gained in realistic verifications are of interest to the automated theorem proving community and new theorem proving techniques should be transferred into practice. The overall objective of the VERIFY workshops is to identify open problems and to discuss possible solutions under the theme "What are the verification problems? What are the deduction techniques?".

This volume contains the research papers presented at the *5th International Verification Workshop* (VERIFY'08) held August 10–11, 2008 in Sydney, Australia. This workshop was the fifth in a series of international meetings since 2002. It was affiliated with the *4th International Joint Conference on Automated Reasoning* (IJCAR 2008).

Each paper submitted to the workshop was reviewed by three referees, and an intensive discussion on the borderline papers was held during the online meeting of the Program Committee. 7 research papers were accepted based on originality, technical soundness, presentation, and relevance. We wish to sincerely thank all the authors who submitted their work for consideration. And we would like to thank the Program Committee members and other referees for their great effort and professional work in the review and selection process. Their names are listed on the following pages.

In addition to the contributed papers, the program included two excellent keynote talks. We are grateful to Prof. Gilles Barthe (IMDEA Software, Madrid, Spain) and Prof. Gernot Heiser (National ICT and Univ. of New South Wales, Sydney, Australia) for accepting the invitation to address the workshop.

August 2008

Bernhard Beckert
Gerwin Klein

Program Co-Chairs and Organisers

Bernhard Beckert	University of Koblenz-Landau, Germany
Gerwin Klein	National ICT Australia, Sydney, Australia

Program Committee

Serge Autexier	DFKI and University Saarbrücken, Germany
Gilles Barthe	IMDEA Software, Madrid, Spain
Peter Baumgartner	National ICT Australia, Canberra, Australia
Bruno Dutertre	SRI International, USA
Reiner Hähnle	Chalmers University, Gothenburg, Sweden
Andrew Ireland	Heriot-Watt University, Edinburgh, UK
Joseph Kiniry	University Dublin, Ireland
Heiko Mantel	TU Darmstadt, Germany
Stephan Merz	INRIA Lorraine, France
Carroll Morgan	Univ. of New South Wales, Sydney, Australia
Peter Müller	Microsoft Research, Redmond, USA
Michael Norrish	National ICT Australia, Canberra, Australia
Wolfgang Paul	Saarland University, Saarbrücken, Germany
Lawrence C. Paulson	University of Cambridge, UK
Wolfgang Reif	University of Augsburg, Germany
Wolfram Schulte	Microsoft Research, Redmond, USA
Johann Schumann	NASA Ames Research Center, USA
Luca Viganò	University of Verona, Italy
Toby Walsh	National ICT Australia, Sydney, Australia
Christoph Walther	TU Darmstadt, Germany

Steering Committee

Serge Autexier	DFKI and University Saarbrücken, Germany
Heiko Mantel	TU Darmstadt, Germany

Additional Referees

Burkhard Wolff
Cesare Tinelli
Peter H. Schmitt
Simon Bäumler

Table of Contents

Invited Talks

Research Papers

Certificate Translation*

Gilles Barthe

IMDEA Software, Madrid, Spain
gilles.barthe@imdea.org

Abstract

Program verification techniques based on programming logics and verification condition generators provide a powerful means to reason about programs. Whereas these techniques have very often been employed in the context of high-level languages in order to benefit from their structural nature, it is often required, especially in the context of mobile code, to prove the correctness of compiled programs. Thus it is highly desirable to have a means of bringing the benefits of source code verification to code consumers.

Certificate translation is a general method to transfer to code consumers evidence gained through verification of source code; it relies on the notion of certificate, used in Proof-Carrying Code to convey to the code consumer independently verifiable evidence that programs respect policies. The talk provides sufficient conditions of existence for algorithms that transform certificates of source programs into certificates of compiled programs, and show that many common transformations comply with these conditions.

* Joint work with Benjamin Grégoire, César Kunz, and Tamara Rezk

Operating System Verification for Real Use

Gernot Heiser

School of Computer Science and Engineering, University of New South Wales, and
Embedded, Real-Time and Operating Systems Program, National ICT Australia
Sydney, Australia
gernot@nicta.com.au

Abstract

Software verification remains an academic exercise as long as it focusses on toy problems, such as systems that are too simplified for practical deployment, or perform too poorly. Furthermore, formal verification of software is of limited benefit if the software is deployed in a system where it executes on top of an unverified operating system.

This talk presents an overview of an effort at NICTA which aims to formally verify a complete operating-system kernel, designed for deployment in mainstream embedded systems. It will explain the approach taken to address the conflicting goals of verifiability, general applicability and high performance. The kernel, called seL4, is designed to replace commercially-deployed high-performance L4 microkernels with no more than 10% performance degradation. The project, which has been running since early 2004, is scheduled to complete by the end of this year.

Model Checking for Stability Analysis in Rely-Guarantee Proofs

Hasan Amjad and Richard Bornat

Middlesex University School of Computing Science, London NW4 4BT, UK
Hasan.Amjad@cl.cam.ac.uk R.Bornat@mdx.ac.uk

Abstract. Rely-guarantee (RG) reasoning is useful for modular Hoare-style proofs of concurrent programs. However, RG requires that assertions be proved stable under the actions of the environment. We cast stability analysis as a model checking problem and show how this may be of use in interactive and automatic verification.

1 Introduction

Multi-core and multi-processor computing systems are now mainstream. Consequently, concurrent programs are the focus of much recent research on automatically proving safety, correctness and liveness properties. Often, the assertions we would like to prove are not amenable to existing automatic analyses. This paper studies one such scenario, and shows how existing automatic techniques can nonetheless help the proof process. The demonstration is expected to be the first step towards a fully automatic method.

Shared-memory concurrency, where multiple threads have read/write access to the same memory addresses, is commonplace. The main challenge in proving properties of such programs, and indeed in their design, is dealing with interference, i.e., the possibility that threads may concurrently make changes to the same memory address.

The concurrent programming community has evolved several synchronisation schemes to avoid interference. Most rely on some form of access denial, such as locks. Whereas locks make it easy to reason about the correctness, they may also cause loss of efficiency as threads wait to acquire locks on needed resources. Locking schemes have thus become increasingly fine-grained, attempting to deny access to the smallest possible size of resource, to minimise waiting and maximise concurrency. The ultimate form of such fine-grained concurrency are programs that manage without any synchronisation at all [14].

The finer the concurrency, the more involved the logic for avoiding interference. This logic must implicitly or explicitly take the actions of other threads into account. This is a problem for program proofs where we strive for modularity, i.e., we wish to be able to reason about a piece of code in isolation from the various environments it could execute in.

Rely-guarantee reasoning [11] offers a solution to this problem within the framework of Hoare-style program proofs [10], by encoding the environment into

the proof: all assertions must be shown to be unaffected under the actions of the environment. Automatically checking for and ensuring such non-interference can be problematic in many cases. In this paper, we describe preliminary progress on a possible solution.

The next section gives brief relevant background. We then describe our method, and comment on shortcomings and possible developments. We assume some familiarity with program proofs using Hoare logic [10], and with model checking [2].

2 Preliminaries

2.1 Rely-guarantee Reasoning

Rely-guarantee (RG) is a compositional verification method for shared memory concurrency introduced by Jones [11]. Interference between threads is described using binary relations. In that treatment, post-conditions were relational, so assertions could talk about the state before and after an action. Here, in line with traditional Hoare logic, we shall use post-conditions of a single state, as this usually makes for simpler proofs. In either case, the essence of RG is unaffected by this choice.

Our command language will be the one used by Jones [11], i.e., with assignment, looping, branching, sequential composition and parallel composition, using C-like syntax. For parallel composition we assume standard interleaved execution semantics, i.e., threads are programs with access to some shared state, and atomic instructions occur interleaved.

Program variables will range over $\mathbb{B}$ and $\mathbb{N}$. It may seem odd to have program variables range over infinite types. In practice however, reasoning about numbers with the aid of abstraction, has been found to be more tractable than reasoning about finite but huge state spaces over words or bit-vectors, which are harder to abstract due to fiddly problems with overflow and underflow.

RG can be seen as a compositional version of the Owicki-Gries method [15]. The specification for a command C is a four-tuple (P, R, G, Q), where P and Q are the usual Hoare logic pre- and post-condition assertions on a single state. C satisfies this specification if from a state satisfying P, and under environmental interference R (the *rely*), C causes interference at most G (the *guarantee*), and if it terminates, it does so in a state satisfying Q.

R and G summarise the properties of the individual atomic actions invoked by the environment and the thread respectively. An action is given as a binary relation on the shared state, and is written $P \rightsquigarrow Q$. This notation indicates that the action updates the part of shared state that satisfies P (at the moment the action executes), so that it satisfies Q.

For example, the action corresponding to the command `x := x + 1`, that increments a shared integer x, might be written as

$$x = N \rightsquigarrow x = N + 1$$

where the implicitly existentially quantified N serves to relate the state before and after the execution. Such *logical* variables are required for describing actions using single-state assertions. We shall denote them using $N, M, \ldots$ and assume they are existentially quantified with scope limited to the action.

G is the relation given by the reflexive and transitive closure of all actions of the thread being specified. The actions are given by manual annotation, as in general, automatic action discovery is non-trivial. R is calculated in an identical manner from the actions of the environment. Typically, the actions comprising R are just the G actions of all the other threads.

An assertion P on a single state is considered *stable* under interference from a binary relation R if $(P; R) \Rightarrow P$, i.e., if $P(s)$ and $(s, s') \in R$, then $P(s')$. More specifically, if P is the pre-condition for some command C, then it must continue to hold after any environment action, before the execution of C. For our purpose, we do not need to pin down the level of atomicity of execution.

Jones gives a full proof system for the satisfaction relation, but we will not need it for this work. However, we reproduce the two critical rules here, to make our assumptions about RG concrete. The first rule is parallel composition, where $||$ is the interleaving parallel composition operator.

$$\frac{(P_1, R \vee G_2, G_1, Q_1) \vDash C_1 \qquad (P_2, R \vee G_1, G_2, Q_2) \vDash C_2}{(P_1 \wedge P_2, R, G_1 \vee G_2, Q_1 \wedge Q_2) \vDash C_1 || C_2}$$

The second rule tells us what it means for a command to be atomic.

$$\frac{(P, id, \mathbf{true}, Q \wedge G) \vDash C \qquad P \text{ stable under } R}{(P, R, G, Q) \vDash atomic(C)}$$

Note one departure from standard RG: the post-condition of the very last line of code is not checked for stability. It is instantaneously true immediately after execution of that line. At this point, either the thread terminates, so that we do not care whether the environment interferes with the post-condition, or, the thread resumes execution from some command the pre-condition of which will be the same as this post-condition, and thus will be checked for stability.

2.2 Temporal Logic Model Checking

Let V be the set of program variables (or *state variables*) used in a program (with appropriate scope management, which we ignore without loss of generality).

Each $v \in V$ ranges over a non-empty set of values D_v. The state space S of the program is given by $\prod_{v \in V} D_v$. A single state of the program is then a value assignment to each $v \in V$.

Suppose AP is the set of all those atomic propositions over V that we might use in the specification of a program. Then we can turn the program into a state machine (technically, a Kripke structure) M represented as a tuple $(S, S0, T, L)$ where S is the set of states, $S0 \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation, and $L : S \to 2^{AP}$ labels each state with the subset of AP that is true in that state.

A temporal logic augments propositional logic with modal and fix-point operators. The semantics of a temporal logic formula in which the atomic propositions range over AP can be expressed in terms of sets of states and/or sequences of states of M. If we turn a program into a state machine, we can use temporal logics to express time-dependent properties of the program.

The most common such property is the global invariant, i.e., a property that holds in all states of a state machine, or equivalently, always holds during the execution of a program.

Global invariants can be checked automatically using proof procedures known as model checkers, subject only to time and space constraints. More importantly, if the proof attempt fails, the model checker can return a counterexample, which is an execution path (sequence of states) leading from an initial state to a state in which the invariant is not satisfied.

The problem of model checking global invariants is in general undecidable when the state space is infinite. However, the ability to produce counterexamples has led to the development of counterexample guided abstraction refinement (CEGAR) [3, 16], where the state space is first abstracted to a simpler one, and if the constructed abstraction is too general it can often be automatically iteratively refined until the desired property is verified. For our purposes we will assume a simple abstraction scheme consisting of a single total *abstraction function* $\alpha : S \to A$, where A is the abstract state space (the exact structure of which depends on the α under consideration). Typically, α is not injective and need not be surjective.

We do not need to describe model checking or CEGAR in more depth, particularly as there are many different abstraction schemes and CEGAR techniques. Further details may be found in [2, 3, 7, 16].

3 Stability Analysis as Model Checking

If the assertion permits, stability can be checked syntactically and unstable assertions can be automatically stabilised by a fix-point computation that disjunctively adds state until stability is achieved. More precisely, given an assertion

satisfying a set of states s, we compute the fix-point by

$$s_0 = s \qquad s_{n+1} = s_n \cup R(s_n)$$

until $s_{n+1} = s_n$, i.e, performing n environment transitions. If the domain of any program variable is infinite, this fix-point computation might not terminate. In such cases, automatic stabilisation techniques rely on abstract interpretation to simplify the domain.

As a simple example, consider the assertion $x = 10$ that is clearly unstable under the environment action $x = N \rightsquigarrow x = N + 1$. To stabilise, we would proceed

$$\begin{aligned} s_0 &= (x = 10) \\ s_1 &= (x = 10 \vee x = 11) \\ s_2 &= (x = 10 \vee x = 11 \vee x = 12) \\ &\vdots \end{aligned}$$

so automatic stabilisation will not terminate. To fix it, we could use the boolean abstraction $\alpha(x) \iff x \geq 10$. Under this abstraction the action above becomes the identity action in all cases except when $x = 9$, but in that case $x = 10$ does not hold anyway, so we have stability immediately, and the assertion is stabilised to $x \geq 10$.

In general, if the abstraction is too weak, it may throw away so much information that the proof becomes impossible (e.g., we can trivially stabilise any assertion by replacing it with **true**). But if the abstraction is too strong, the fix-point computation may not terminate, or may run out of time or space resources.

Current techniques therefore use hand-crafted abstraction heuristics that are found to work in practice, for the underlying variable domains [5, 17].

We have seen in §2.2 that this problem of finding exactly the right level of abstraction also occurs in model checking. It is our hope that the model checking solution can be applied to stability analysis as well. If so, the vast amount of model checking research on this topic can be brought to bear on the problem. We now present the first step towards this goal, by representing stability analysis as a model checking problem.

Rather than representing R and G as the reflexive transitive closures of their constituent actions, we consider them as state machines over the shared state. In addition, the state machine also has state variables for the program counters of the constituent threads. Tracking program counters allows us to easily encode the flow control of actions in the state machine.

The state machine for the guarantee condition G_t for a thread t, is $M_t = (S_t, S0_t, t, L_t)$ and is constructed as follows. Let V_t be the set of all shared program variables used in t as well as the t program counter $pc_t : \mathbb{N}$. Then S_t is

constructed like S in §2.2. $S0_t$ is those states of S_t in which $pc_t = 0$ and in which any other $v \in V_t$ are assigned their initial values if any. Let a_l be the (possibly empty) set of actions associated with the primitive command on line l of the thread code (this is for easy specification: in reality a single atomic statement can only have a single associated action, so the analysis simply conjoins them). Then

$$t((s, pc_t), (s', pc'_t)) = \bigvee_l \left(pc_t = l \wedge pc'_t = next(s, l) \wedge \bigwedge_{a \in a_l} a(s, s') \right)$$

where the *next* function encodes the control flow of thread t. Finally, $L_t(s) = \{p \in AP \mid p(s) = \mathbf{true}\}$. The state machine for R, $M_R = (S_R, S0_R, T_R, L_R)$ over variables V_R, is constructed analogously, though of course it is more complicated since it encompasses the actions of all the other threads.

Now suppose that we wish to stabilise an assertion P that is the pre-condition of a command at program line l in a thread t. The first step is to check whether the assertion is already stable. To do this we augment M_R with fresh[1] variables corresponding to any thread-local variables that occur in P, and also add identity actions over these variables to T_R so that their values never change. This represents our intuition that when checking the stability under the environment of an assertion in thread t, t itself is not executing.

We can now model check the augmented state machine M'_R for the global invariant P. Note that here we can use standard model checking abstraction construction techniques [7] to try and avoid non-termination. If the invariant holds, then since it is a global invariant and α is total, it holds in the concrete state space as well. Otherwise, we will obtain a counter-example giving a sequence of actions of the abstract state machine that violates the invariant. At this point, standard CEGAR techniques can be employed to check whether the counterexample has a corresponding concrete trace, in which case the stability check has failed. If not, the abstract trace is spurious (caused by too weak an abstraction), so we refine the abstraction using standard CEGAR methods and call the model checker again, until we have success or failure.

At this point we have already improved on existing stabilisation methods by not being reliant on having a syntactic check for stability. However, we still have to handle the case where the stability check fails.

In this situation, we have at hand a counterexample trace π showing a sequence of environment actions that falsified P, and also the particular abstraction function α being used by the model checker when the stability check failed. These two pieces of information can be used to weaken P and then repeat the stability check, and iterate until P is stable.

[1] So that there is no name clash with any $v \in V_R$.

For example, if we are using predicate abstraction techniques, then for our running example where we are checking stability of $P \equiv x = 10$, we may have

$$\alpha(x) \iff x = 10 \quad \text{and} \quad \pi \equiv x = 10 \rightsquigarrow_k x = 11$$

where k uniquely identifies the action responsible. This information suggests generalising P to $x \geq 10$, and then the stability check succeeds.

This is as far as we have come. The new weakened assertion must be found manually for now, using the point-of-failure α and π as guides, as in the example above. Of course, we could simply use the existing method of repeated disjunctive addition of the resultant state of the involved actions (which in this cherry-picked example fails to terminate). However, the model checking approach gives us extra information (point-of-failure π and α) which should hopefully allow us to do better. We plan to develop an automatic method that uses symbolic simulation driven by the counterexample traces, perhaps in combination with heuristics, to weaken P in a useful manner. Here, we expect to use existing model checking research on automatic abstraction construction [8].

In fact, at the moment our in-development tool (effectively a translation layer on top of the NuSMV model checker [1]) does not even perform abstraction, as all our test assertions are over finite domains. This is because while it would be simple to switch to a tool supporting automatic abstraction (e.g., BLAST [9]), we are more interested in finding out how to use π to weaken P, which is the real challenge.

3.1 Comment: Refining Stability

In standard RG, the rely is represented as the reflexive transitive closure of all actions that the environment can execute. This can also be thought of as a state machine, albeit a not very informative one in which any transition (action) can execute from any state. Thus, our representation of R and G as state machines of actions can be seen as a refinement of the standard RG representation. The latter can be thought of as the state machine consisting of all states reachable from any state via all possible interleavings of the underlying actions, regardless of whether these interleavings will ever actually occur. Our refinement proceeds by adding control flow information, thus ruling out certain interleavings.

Thus it is possible for us to prove the stability of stronger assertions than is possible in standard RG. Since the stability check is orthogonal to the RG proof system, this means that we automatically obtain a stronger proof system.

Indeed, we can parameterise the RG proof system by the level of refinement of R and G. We have experimented along these lines by adding some G actions to R, or by selectively exposing the thread-local state of the environment, both of which rule out some class of impossible action sequences. In each case we

have been able to prove properties that are stronger, and often more intuitive to specify.

There is a trade-off here, since adding more information to R and G will almost certainly make the underlying model checking problem harder, affecting scalability. Nonetheless, increasing the refinement level is attractive not only because it permits stronger properties to be proved, but also because the stabilised assertion may be syntactically smaller and thus more readable. This latter consideration is important if these methods are used as part of a larger interactive proof framework, such as a theorem prover.

4 Remarks

We do not know of any other work that uses model checking for stability analysis in Hoare-style RG proofs. There is work underway at MSR Cambridge [6] that also represents R and G as state machines, but their aim is to deal with questions of liveness. Other than that we know only of the automatic stabilisation work that inspired our own effort [17].

It is well known [4] that the standard RG proof rule for parallel composition can become unsound if the satisfaction relation is strengthened (e.g., to include liveness). We are safe since stability is a safety property, for which the standard RG proof system is sound.

Apart from the unfinished aspect, this approach has other shortcomings. An important one is that refining R and G quickly makes the underlying model checking problem more resource intensive. The same refinement (specifically, the need to track program counters) also prevents the results from scaling up to arbitrary numbers of threads for free, unlike in standard RG. We expect that model checking techniques like paramatric verification [13] and assume-guarantee reasoning [12] (not to be confused with rely-guarantee) may help with this. More generally, our ability to change the refinement level of R and G should also help ameliorate the situation.

We are also considering the use of separation logic in this framework, to frame out irrelevant state and thus alleviate our model checking woes. RG and separation logic have already been combined [18]. Extending that framework to our method will be another thread of future work.

Acknowledgement The first author would like to thank Viktor Vafeiades for permission to copy from the description of RG in his Ph.D. thesis.

References

1. Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveriand, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *LNCS*. Springer, March 2002.

2. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
3. E. M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification - (CAV'00)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
4. Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. CUP, 2001.
5. Dino Distefano, Peter O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.
6. Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving liveness properties of non-blocking data structures. Submitted to POPL 2008.
7. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, June 1997.
8. Arie Gurfinkel, Ou Wei, and Marsha Chechik. Systematic construction of abstractions for model-checking. In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 381–397. Springer, 2006.
9. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 262–274. Springer, 2003.
10. C. A. R. Hoare. An axiomatic basis for programming. *Communications of the ACM*, 12(10):576–580, 1969.
11. Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
12. K. L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *LNCS*, pages 219–234. Springer, 1999.
13. K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In Tiziana Margaria and Thomas F. Melham, editors, *Proceedings of the 11th International Conference on Correct Hardware Design and Verification Methods*, volume 2144 of *LNCS*, pages 179–195. Springer, 2001.
14. Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM Press, 1996.
15. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
16. H. Saïdi. Model checking guided abstraction and analysis. In Jens Palsberg, editor, *Proceedings of the 7th International Static Analysis Symposium*, volume 1824 of *LNCS*, pages 377–396. Springer, July 2000.
17. Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.
18. Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*, volume 4037 of *LNCS*, pages 256–271, 2007.

Compositional Proofs with Symbolic Execution

Simon Bäumler, Florian Nafz, Michael Balser, and Wolfgang Reif

Institut für Informatik – University of Augsburg
Augsburg, Germany

Abstract. A proof method is described which combines compositional proofs of interleaved parallel programs with the intuitive and highly automatic strategy of symbolic execution. As logic we use an extended variant of Interval Temporal Logic that allows to formulate programs directly in the Simple Programming Language (SPL). The notation includes a complex interleaving operator. The interactive proof method we use for temporal properties is symbolic execution with induction. Here, we show how to combine this proof method with an assumption-guarantee approach to decompose proofs for safety properties. We demonstrate the application of this technique with a producer-channel-consumer case study. [1]

1 Introduction

Verification of concurrent systems is an important topic, as, in comparison to sequential programs, the system execution is much more complex. Validation of concurrent systems by testing is very difficult and often not feasible, as there are many more test cases and it is hard to reproduce tests. But also formal verification of concurrent systems is complicated, because reasoning over all possible execution traces tends to result in a huge state space which makes automatic and interactive verification very difficult.

To avoid reasoning over the complete concurrent system, a common technique is compositional reasoning. The basic idea of this technique is, to split the system into several subcomponents. Then, the overall property is proved only with corresponding properties of the subcomponents. This idea was first formulated by Dijkstra [1]. In compositional reasoning the proof is often done with a compositional theorem. Such a theorem provides a number of proof obligations, which have to be fulfilled, so that the overall property is valid. Ideally, these proof obligations contain only single subcomponents and properties of these subcomponents, but not the complete system itself. This results in several proofs of feasible size.

A common compositional proof technique is the assumption-guarantee paradigm, which was introduced by Jones [2] and by Misra & Chandy [3]. The basic idea of this paradigm is, that each component can make specific assumptions to its environment in order to guarantee a specific behavior. An overview of recent works on compositionality in general can be found e.g. in de Roever et. al. [4] or Furia [5].

[1] This work has been funded by the DFG program INOPSYS II, under contract number Re 828/6-3.

Symbolic execution, on the other hand, is a successful technique for interactive verification of sequential programs (e.g. Dynamic Logic [6, 7]). It is a very intuitive strategy for programs as the proof advances step by step similar as most humans do it when trying to understand a program [8, 9]. Furthermore, it can be automated to a large extend. Balser [10] presented an ITL[2]-based logic with calculus that allows the symbolic execution of concurrent systems. This calculus was integrated into the interactive theorem prover KIV [12]. Arbitrary specification languages can be nested into this logic and thus making it unnecessary to translate a system specification into a special specification language for formal verification. Even more important is that the interleaving in this logic is compositional. That means, it is possible to replace a subcomponent with an abstraction of the component in a concurrent proof. While this feature simplifies concurrent proofs, it is still necessary to use symbolic execution on the whole parallel system in order to prove a property. A compositional theorem for this method would make it possible to prove properties of concurrent systems by reasoning only over single subcomponents at a time.

The goal of this paper is to present an assumption-guarantee rule for the logic presented in [10]. This would enable us to fully use the advantages of both techniques, compositional reasoning and symbolic execution, as well as the tool support, which is available for this logic.

We assume that the reader has at least basic knowledge in temporal logic and sequent calculus. The remainder of the paper is structured as follows: A short overview of our logic is given in Section 2. The compositional theorem we use is presented in Section 3, its application is shown in Section 4 on a producer-channel-consumer case study. Section 5 concludes the paper with related work and an outlook.

2 Temporal Logic Framework

In the following an informal overview over the used temporal logic calculus is given. The formal semantic is described in [13] and [10]. The calculus is integrated into the interactive theorem prover KIV. The temporal logic framework is a variant of ITL [11, 14] that is extended by explicitly including the behavior of the environment into each step. The basis for ITL are finite or infinite sequences π of valuations, which are called *intervals*. Valuations in π are called *states*. Each state is described by a first-order predicate logic formula over dynamic variables v, which also can be *primed* v' or *double primed* v''. The relation between v and v' is called *system transition*, whereas the relation between v' and v'' *environment transition*. The value of v'' in a state must be equal to the value of v in the next successive state. Thereby the system and the environment transition alternate. A selection of the supported temporal operators are:

[2] Interval Temporal Logic, introduced by Moszkowski [11]

$\circ\varphi$	there is a *next* state and it satisfies φ
last	the current state is the last state
$\Box\varphi$	φ holds *always* from now on in every state
φ **unless** ψ	either φ holds always from now on in every state or ψ holds in any state and φ holds in every state before
$\lceil \boldsymbol{v} \rceil$	*frame assumption*, only variables in $\boldsymbol{v}$ are modified
$\varphi_1 \parallel \varphi_2$	interleaving

Further, programs are written in a SPL (Simple Programming Language) [15] like program syntax. The selection of the used SPL-operators are:

$x := t$	assignment	**await** ψ	synchronization
$\varphi_1 \vert \varphi_2$	parallel assignment	**while** ψ **do** φ	loop
$\varphi_1; \varphi_2$	sequential composition		

Semantically, a program describes a set of traces. Therefore, it is possible to embed programs into temporal formulas. This can be used for the parallel composition of programs with the tl-interleaving operator.

2.1 Symbolic Execution

A typical sequent in proofs about interleaved programs has the form $P, \Gamma \vdash \Delta$. Here, P is the interleaved program, Γ contains a temporal formula that describes the environment behavior and a first order formula for the current variable assignment, while Δ contains the temporal property which has to be shown.

Symbolic execution on the following example sequent is done in two steps:

$$m := m + 1; < \text{prog} >,\ \Box m' = m'', m = 2 \ \vdash\ \Delta$$

First, all temporal and program formulas are rewritten to a so called first-next form, which encodes the transition to the next state in a predicate logic formula. For this, the following rule[3] is used:

$$\frac{m' = m + 1,\ \circ < \text{prog} >,\ m' = m'',\ \circ\Box m' = m'',\ m = 2 \ \vdash\ \Delta}{m := m + 1; < \text{prog} >,\ \Box m' = m'',\ m = 2 \ \vdash\ \Delta} \quad \text{(prenex)}$$

This rule separates propositons about the current state from propositions about all following states. So after application of *prenex* each formula is either a first-order formula, describing the first state in the trace or a temporal formula with a leading *next*-Operator, that describes the remaining trace.

Now it is possible to advance one step in the trace. In all first-order formulas, unprimed and primed variables are replaced by new static variables, while the double primed variables are replaced by their unprimed version. Further, all

[3] Note that rules in the sequent calculus are read bottom-up, with the conclusion at the bottom and the corresponding proof obligations on the top part.

next-operators of temporal formulas are eliminated. In the example, this is done by the following rule-application:

$$\frac{M_1 = M_0 + 1,\ <\text{prog}>,\ M_1 = m',\ \Box m' = m'',\ M_0 = 2\ \vdash\ \Delta}{m' = m + 1,\ \circ <\text{prog}>,\ m' = m'',\ \circ\Box m' = m'',\ m = 2\ \vdash\ \Delta} \qquad \text{(tl-step)}$$

This results in the following sequent after simplification:

$$<\text{prog}>,\ \Box m' = m'',\ m = 3\ \vdash\ \Delta$$

The rules for symbolic execution of formulas in the succedent are very similar. In KIV these rules, *prenex*, *tl-step* and simplification, are combined to a single complex rule called *step*.

2.2 Executing Interleaved Programs

To execute two interleaved formulas a first transition from one or the other formula is executed. After this, execution continues with interleaving the remaining formulas. For example, if there are two interleaved programs in the antecedent $m := 1; \ldots \parallel n := 2; \ldots,\ \Gamma\ \vdash\ \Delta$ this formula is executed by symbolically executing either program first. For this, the following rule is used:

$$\frac{\begin{array}{l} m := 1; (\ldots \parallel n := 2; \ldots),\ \Gamma\ \vdash\ \Delta \\ n := 2; (m := 1; \ldots \parallel \ldots),\ \Gamma\ \vdash\ \Delta \end{array}}{m := 1; \ldots \parallel n := 2; \ldots,\ \Gamma\ \vdash\ \Delta} \qquad \text{(interleaved left)}$$

Furthermore the following equation holds for the interleaving operator

$$\textbf{last} \parallel \phi \quad \leftrightarrow \quad \phi$$

which can be used to eliminate terminated programs. In the case that one of the programs is blocked, only the other program is executed.

One important feature of our interleaving operator is that it is compositional. This means, that the following rule can be applied:

$$\frac{\vdash\ \varphi_1 \rightarrow \varphi_2 \qquad \varphi_2 \parallel \psi,\ \Gamma\ \vdash\ \Delta}{\varphi_1 \parallel \psi,\ \Gamma\ \vdash\ \Delta} \qquad \text{(comp)}$$

This feature is very important for the proofs of the theorems in chapter 3 and for abstraction in general.

Note, that our interleaving operator also supports features like fairness and blocking. These features and the general case, where the interleaving operator contains arbitrary temporal formulas, are also described in detail in [16] or [10].

2.3 Induction and Sequencing

The basic idea to proof safety properties is to advance in the interval until a valuation is reached that was considered earlier in the interval, so that a loop was executed. If it can be proven that the property is true before and during the loop so it is invariant, then the proof can be finished with an inductive argument. A special rule *start induction* is used to generate a suitable induction hypothesis.

Symbolic execution can lead to many paths, that have to be explored. Often two different paths lead to same configurations (two sequent have the same configuration if all temporal logic formulas are the same). To minimize the proof effort a rule called sequencing is used, that allows to close a open premise when there exists another premise with the same configuration, but with more general predicate logic formulas.

3 Compositional Theorem

Most assumption/guarantee based compositional proof techniques use a special operator similar to the "while-plus" operator $\overset{+}{\rightarrow}$ presented in [17]. Informally, the term $A \overset{+}{\rightarrow} G$ means, that if A holds up to step i, then G must hold up to step $i+1$. This operator enables the formulation that a component violates its guarantee G only after its assumption A is violated. It is needed to break the circularity of the used compositional rule.

Assumptions and guarantees can be formulated with propositional predicates over unprimed and primed variables (e.g. Cau and Collette [18]). We use the same approach, but for the assumptions we use predicates over primed and doubly primed variables. In this way it can be formalized which steps are allowed for the components and which steps are allowed for the environment. This also allows to use a standard TL operator **unless** as $\overset{+}{\rightarrow}$ operator, i.e.:

$$A \overset{+}{\rightarrow} G := G \,\mathbf{unless}\, (G \wedge \neg A)$$

With these preliminaries we are able to construct a compositional theorem:

Theorem 1. *If:*

i. *for all* $i = 1, \ldots, n$: $M_i \vdash A_i(v', v'') \overset{+}{\rightarrow} G_i(v, v')$
ii. *for all* $i = 1, \ldots, n$: $G_i(v_1, v_2) \vdash G(v_1, v_2) \wedge \bigwedge_{j \in \{1..n\} \wedge j \neq i} A_j(v_1, v_2)$
iii. *for all* $i = 1, \ldots, n$: $A_i(v_1, v_2) \wedge A_i(v_2, v_3) \vdash A_i(v_1, v_3)$
iv. $A(v_1, v_2) \vdash \bigwedge_{i \in \{1..n\}} A_i(v_1, v_2)$

then:

$M_1 \parallel \ldots \parallel M_n \vdash A(v', v'') \overset{+}{\rightarrow} G(v, v')$

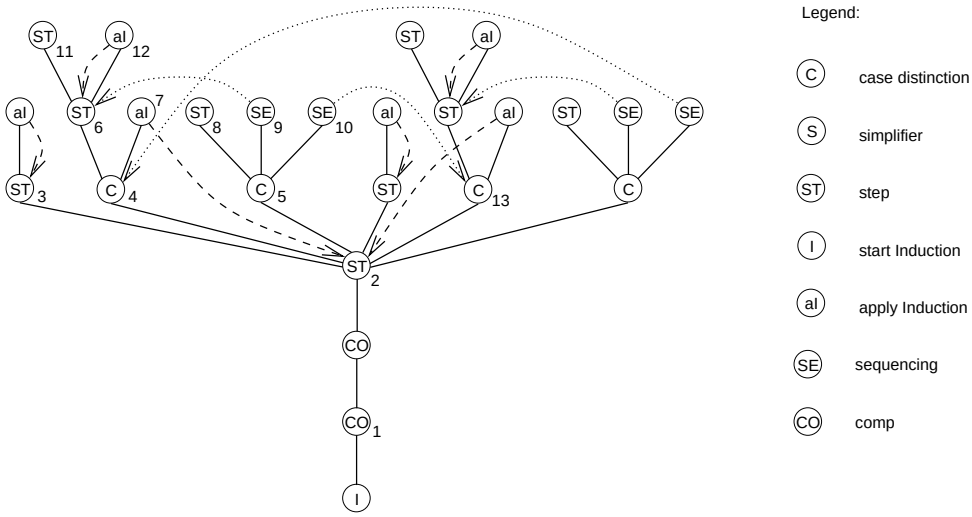


Fig. 1. Proof Graph for Theorem 1

Premise *i* is a temporal logic sequent while premise *ii* - *iv* contain only predicate logic formulas. These four proof obligations have the following informal meaning:

i. All components must sustain their guarantee as long as the assumption holds. These are the only proof obligations which require a temporal logic proof.
ii. The guarantee of each component preserves the global guarantee and does not violate the assumptions of all other components.
iii. The assumptions of all components are transitive. With this property, the components assumption is preserved even if other components make several steps.
iv. All component assumptions hold if the global assumption holds. Therefore, no component assumption is violated in the environment-step.

Proof (Outline).

The theorem was formally proven with the theorem prover KIV by using the ITL calculus described in section 2. As first step the proof for two components was done by symbolic execution of two abstract and interleaved components. The simplified proof graph[4] for this first step is depicted in figure 1. The premises *i-iv* of theorem 1 are used as lemmas for this proof. Premises *ii-iv* are applied by the KIV simplifier on predicate logic premises, which are all closed automatically by KIV. These simplifier steps are omitted in figure 1 for the sake of brevity.

The proof starts with the sequent $M_1 \parallel M_2 \vdash A(v', v'') \overset{+}{\rightarrow\!\triangleright} G(v, v')$ (node 1). M_1 and M_2 are abstract programs that have arbitrary behavior. At first both

[4] The rules *apply Induction* and *Sequencing* refer both to another node in the proof tree, as explained in section 2. Therefore we depict proofs as graphs. The nodes that are referred by the rules *apply Induction* and *Sequencing* are represented by dashed arrows and pointed arrows respectively.

programs are replaced with their assumption-guarantee formulas of premise i of the theorem via the rule *comp*, so node 2 has the following sequent: $A_1(v', v'') \overset{+}{\rightarrow} G_1(v, v') \parallel A_2(v', v'') \overset{+}{\rightarrow} G_2(v, v') \vdash A(v', v'') \overset{+}{\rightarrow} G(v, v')$ Here, the *step* rule is applied for symbolic execution. In the following, only the nodes 3-5 are described, as the other three premises of node 2 are symmetrical to these nodes.

In node 3 the first parallel component has terminated, so it must be shown that $A_2(v', v'') \overset{+}{\rightarrow} G_2(v, v') \vdash A(v', v'') \overset{+}{\rightarrow} G(v, v')$ holds. This can by done using *step* and *apply induction*.

In node 4 the first component has made a normal step (i.e. it is neither terminated nor blocked). The case distinction discerns if $A_1(v', v'')$ holds (node 7) in this step or not (node 6). Node 6 has the sequent $\neg A_1(v', v'') \parallel A_2(v', v'') \overset{+}{\rightarrow} G_2(v, v') \vdash A(v', v'') \overset{+}{\rightarrow} G(v, v')$. Here, in the next step there are three possibilities:

- The left component makes a step (not depicted in the graph, as it can be closed automatically by the simplifier).
- The right component makes a step and $A_2(v', v'')$ is violated too (node 11). This can be closed automatically by another step.
- The right component makes a step and $A_2(v', v'')$ holds (node 12). This premise can be closed by induction.

Node 7 contains exactly the same sequence as node 2, therefore induction can be applied.

Node 5 treats the case if the left component is blocked. Here, three cases are possible:

- Both assumptions $A_1(v', v'')$ and $A_2(v', v'')$ are violated (node 8). This can be closed automatically via step rule.
- Only the assumption $A_1(v', v'')$ is violated (node 9). This is the same case as in node 6, therefore sequencing can be applied.
- $A_1(v', v'')$ holds and the right component has made a step (node 10). This case is covered in node 13, therefore sequencing can be applied.

This proof can be extended to n components by induction over the number of components. The initial induction case for one component can be shown by another temporal induction (similar to node 3 in the proof above). The inductive step can be proved by using the proof for two components as lemma to reduce n components to $n-1$ components. Then the induction hypotheses can be applied.

Usually the construction of a modularization rule is very difficult because of mutual dependencies. One interesting thing in our framework is that symbolic execution and tool support can not only be used to prove the modularization theorem, it actually helps to find the correct premises for the rule. To do so, the proof is as above, but without using the premises ii-iv as lemmas (as we

want to find them at this point). Then we try to close all open premises that contain temporal logic formulas, which results in a similar proof graph as shown in figure 1, but with several additional open premises that contain only predicate logic sequents. So to find the correct premises for the modularization theorem are a minimal set of generic predicate logic formulas from which all open sequents can be shown. By this technique a semantic analysis of the parallel operator is not necessary.

Extended Modularization Rule While this first rule may be useful for very simple systems it must be improved to be usable for more complex cases. First, a variable initialization in the temporal logic proofs (obligations i) is needed. Second, applications of this first rule show, that the guarantees are often redundant. Especially it is often necessary to have an invariance property. This invariant can be used to express the relation between the initial state and all suceeding states. Similar techniques for these additions are used e.g. in [18].

So, using the additional predicates $I(v)$ for the invariant, $Init(v)$ for the initial values of the global system and a family of predicates $Init_i(v)$ for the initial values for every system component leads to an extended version of the compositional rule:

Theorem 2. *If:*

i. *for all* $i = 1, \ldots, n$*:*
$M_i, I(v), Init_i(v) \vdash A_i(v', v'') \xrightarrow{+} G_i(v, v')$
ii. *for all* $i = 1, \ldots, n$*:*
$G_i(v_1, v_2) \wedge I(v_1) \vdash G(v_1, v_2) \wedge \bigwedge_{j \in \{1..n\} \wedge j \neq i} A_j(v_1, v_2) \wedge I(v_2)$
iii. *for all* $i = 1, \ldots, n$*:*
$A_i(v_1, v_2) \wedge A_i(v_2, v_3) \wedge I(v_1)) \vdash A_i(v_1, v_3)$
iv. $A(v_1, v_2) \wedge I(v_1) \vdash \bigwedge_{i \in \{1..n\}} A_i(v_1, v_2) \wedge I(v_2)$
v. *for all* $i = 1, \ldots, n$*:*
$A_i(v_1, v_2) \wedge I(v_1) \wedge Init_i(v_1) \vdash Init_i(v_2)$
vi. $Init(v_1) \vdash \bigwedge_{i \in \{1..n\}} Init_i(v_1) \wedge I(v_1)$

then:
$M_1 \parallel \ldots \parallel M_n, Init(v) \vdash A(v', v'') \xrightarrow{+} G(v, v')$

The informal meaning of the proof obligation of this theorem are as follows:

i. These obligations are mostly the same, except that we can now assume the invariant and the initial condition for the respective component in the antecedent.
ii. - *iv.* These obligations are mostly the same as in the previous rule, except that the predicate I can now be assumed in the antecedent. Also, we have to show in obligations ii. and iv., that the invariant is preserved by the guarantee of each component and the global assumption.

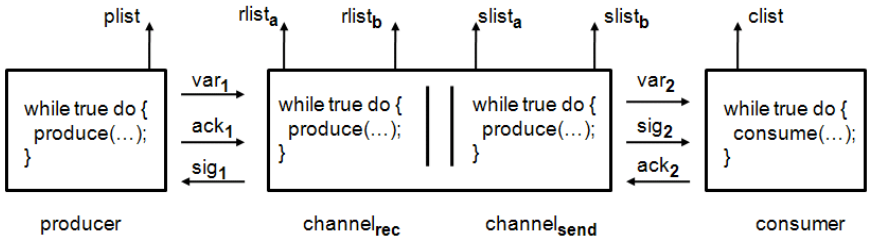


Fig. 2. Producer-Channel-Consumer (ProChaCon)

v. Here it is shown, that the initial condition of a component is preserved by its assumption.
vi. This obligation establishes the invariant and the initial conditions of the components.

Proof (Sketch). This theorem was also formally proven with KIV. The proof for theorem 2 and 1 are very similar. However, it must be shown for theorem 2 that $Init_1(v)$, $Init_2(v)$ and $I(v)$ holds in the first state. To do that, a case distinction is used before the first step (node 2 in figure 1). The cases where one of the formulas $Init_1(v)$, $Init_2(v)$ and $I(v)$ does not hold can be proved via step and induction, similar to node 8 in figure 1.

4 Case Study

In this section an example for applying the introduced theorem is presented. After an introduction of the producer-channel-consumer case study (short "ProChaCon") and its specification the formulation of the assumption-guarantee (short "AG") properties is described. The section closes with a description of the proofs of some of the proof obligations.

ProChaCon consists, as the name implies, of three interleaved components, depicted in Figure 2. Usually the values of the *producer* component are derived from an application or another component. For our task it is sufficient to generate them randomly. These values are sent using a classical two-way-handshake protocol [19] to the *channel* component. The *channel* is again divided into a receiver and a sender component. Both are connected through a buffer in which the incoming values are stored. The receiver is responsible to store the incoming values into the buffer and the senders job is to forward the buffered values. Thereto, the receiver attaches the incoming value to the buffer-list and the sender transmit the first value of the list as long as the buffer is not empty. The buffered values are transmitted to the *consumer* component, which processes the received values in an arbitrary way. The history of sent and received values is modeled by inserting history lists on certain points, e.g plist, also depicted in Figure 2. They

```
producer:
begin
 while true do
   await ch_a.sig =ch_a.ack;
   a := [?];
   ch_a:= mkch(a, ch_a.sig, ch_a.ack) ;
   ch_a:= mkch(ch_a.data,
                ¬ ch_a.sig, ch_a.ack)|
    plist := plist + a
end;
```

```
consumer:
begin
 while true do
   await ch_b.sig ≠ch_b.ack;
   b := ch_b.data;
   ch_b := mkch(ch_b.data,
                ch_b.sig,ch_b.ack)|
    clist := clist + b
end;
```

```
channel:
 begin
1 while true do
2    await ch_a.sig ≠ch_a.ack;
3    c := ch_a.data;
4    ch_a := mkch(ch_a.data,
                  ch_a.sig, ¬ch_a.ack)|
      elist_a := elist_a + c ;
5    chbuf := chbuf + c|
      elist_b := elist_b + c
  end;
```

```
begin
 while true do
   await chbuf≠ [];
   d:= chbuf.first|
    chbuf := chbuf.rest|
    slist_a := slist_a + chbuf.first;
   await ch_b.sig = ch_b.ack;
   ch_b := mkch(d, ch_b.sig,ch_b.ack);
   ch_b := mkch(ch_b.data,
                ¬ch_b.sig,ch_b.ack),
   slist_b := slist_b + d
end;
```

Fig. 3. SPL Representation of ProdChaCon

are implemented as atomic assignments attached to the accordant program step. A specification of the components with SPL is shown below in Figure 4.

First some abbreviatory notations are described that will be used in the following. The sets of all used unprimed, primed and doubleprimed variables are denoted with V, V' and V''. As mentioned in the introduction a step consists of a system step and an environment step. In the following it is often necessary to express that a component only change a set of variables L. This is formulated by a frame assumption, which corresponds to the formula

$$\lceil L \rceil :\Leftrightarrow \bigwedge_{w \in V \setminus L} w' = w$$

which states that all program variables except L are unchanged. Here, L is a subset of V. Further, during the environment step some variables are unchanged. This is formulated with the following predicate

$$Unchanged_{env}(L) :\Leftrightarrow \bigwedge_{w \in L} w' = w''.$$

The verified property is "The list of received values is always a prefix of the list of the sent values". In other words, only values that have been sent are received and the order is unchanged. So for the overall guarantee the formula

$clist \sqsubseteq plist \rightarrow clist' \sqsubseteq plist'$ is used, where $\sqsubseteq$ is the prefix operator. The global assumption states that all variables are unchanged by the environment. This leads to the following proof obligation for the complete system.

$$Unchanged_{env}(V) \xrightarrow{+} (\text{clist} \sqsubseteq \text{plist} \rightarrow \text{clist}' \sqsubseteq \text{plist}')$$

The system uses, as mentioned, a classical handshake to transmit values. Therefore, the involved components have to guarantee at least that they fulfill their part accurate. Sending components must guarantee that they transmit a value only if it is their turn and that the history-lists are updated in a correct way, formally expressed in $Handshake_{send}$.

$$\begin{aligned} &Handshake_{send}(\text{ch, hlist}) :\Leftrightarrow \\ &\quad (\text{ch}.sig \neq \text{ch}.ack \rightarrow (\text{ch} = \text{ch}' \wedge \text{hlist} = \text{hlist}')) \\ &\wedge\ (\text{ch}.sig = \text{ch}.ack \wedge \text{ch}'.sig = \text{ch}'.ack) \rightarrow \text{hlist} = \text{hlist}') \\ &\wedge\ (\text{ch}.sig = \text{ch}.ack \wedge \text{ch}'.sig \neq \text{ch}'.ack) \rightarrow \text{hlist} + \text{ch}.data = \text{hlist}' \end{aligned}$$

Analogously $Handshake_{receive}$ express that the receiver has to guarantee that values are only received if the handshake variables are unequal. The history list is updated if the handshake variables signalizing that a value was received successfully and the next value can be transmitted.

The producer component has to guarantee two things. First, that only internal variables and the handshake channel are changed. Second, that the handshake protocol is implemented correctly. The producers environment assumption A_1 states that the environment does not change the internal variables as long as the producer could transmit a value. This is captured in G_1 and A_1.

$$\begin{aligned} G_1(V, V') &:\Leftrightarrow \lceil\ \text{a,ch}_a\text{,plist} \rceil \wedge\ Handshake_{send}(\text{ch}_a\text{,plist}) \\ A_1(V', V'') &:\Leftrightarrow Unchanged_{env}(\text{a,plist}) \wedge\ (\text{ch}'_a.\text{sig} = \text{ch}'_a.\text{ack} \rightarrow \text{ch}'_a = \text{ch}''_a)) \end{aligned}$$

The AG of the consumer can be formalized analogously:

$$\begin{aligned} G_4(V, V') &:\Leftrightarrow \lceil\ \text{b, ch}_b\text{,clist} \rceil \wedge\ Handshake_{receive}(ch_b, clist) \\ A_4(V', V'') &:\Leftrightarrow Unchanged_{env}(\text{b,clist}) \wedge\ (\text{ch}'_b.\text{sig} \neq \text{ch}'_b.\text{ack} \rightarrow \text{ch}'_b = \text{ch}''_b)) \end{aligned}$$

In a similar way the AGs for both channel components (channel_{rec}, channel_{send}) can be formalized. They need additional guarantees, because they pass the values via a buffer. That this is done correctly is formalized by the two guarantees $Buffer_{in}$ and $Buffer_{out}$.

$$\begin{aligned} &Buffer_{in}(\text{buffer, hlist}_{in}\text{, value}_{in}) :\Leftrightarrow \\ &\qquad (\text{hlist}_{in} = \text{hlist}'_{in} \wedge \text{buffer} = \text{buffer}') \\ &\quad \vee\ (\quad \text{hlist}_{in} + \text{value}_{in} = \text{hlist}'_{in} \\ &\qquad \wedge\ \text{buffer} + \text{value}_{in} = \text{buffer}') \end{aligned}$$

$$\begin{aligned} &\mathit{Buffer}_{out}(\text{buffer}, \text{hlist}_{out}) :\Leftrightarrow \\ &\qquad \text{hlist}_{out} + \text{buffer} = \text{hlist}'_{out} + \text{buffer}' \end{aligned}$$

The complete guarantee for channel_{rec} consists of the statements that channel_{rec} only changes its internal variables, that the receiver part of the handshake protocol is implemented in a correct way and that the component writes into the buffer correctly. Additionally, the component needs to guarantee that the prefix property also holds between both internal history lists. As assumption it can be presumed that the environment does not change the internal variables and the channel is not changed as long as channel_{rec} can receive a value. That leads to the following AG.

$$\begin{aligned} G_2(V, V') \quad :\Leftrightarrow \quad & \lceil \text{c}, \text{ch}_a, \text{chbuf}, \text{rlist}_a, \text{rlist}_b \rceil \\ & \wedge \mathit{Handshake}_{receive}(\text{ch}_a, \text{rlist}_a) \\ & \wedge \mathit{Buffer}_{in}(\text{chbuf}, \text{rlist}_b, \text{c}) \\ & \wedge \text{rlist}'_b \sqsubseteq \text{rlist}'_a \\ A_2(V', V'') \quad :\Leftrightarrow \quad & \mathit{Unchanged}_{env}(\text{c}, \text{rlist}_a, \text{rlist}_b) \\ & \wedge (\text{ch}'_a.\mathit{sig} \neq \text{ch}'_a.\mathit{ack} \ \rightarrow \text{ch}'_a = \text{ch}''_a)) \end{aligned}$$

The AG of the other channel component (channel_{send}) can be formalized analogously with Buffer_{out} and $\mathit{Handshake}_{send}$.

The system always has to be in a correct state. In other words the buffers have to be empty or at least have to be filled in a non-conflicting way. This is expressed as an invariant. Theoretically, it is also possible to put all these into the AGs of the components, but it is more concise to have only local properties there. Therefore statements consisting of variables of more than one component are separated within an invariant, which expresses the connection of the components. First it states that depending on the handshake variables the two neighbor history lists are either equal or they differ in the value that is set in the data field.

$$\begin{aligned} I_1(V) :\Leftrightarrow \ & (\ \text{ch}_a.\text{sig} = \text{ch}_a.\text{ack} \ \rightarrow \text{elist}_a = \text{plist}) \\ & \wedge (\text{ch}_a.\text{sig} \neq \text{ch}_a.\text{ack} \ \rightarrow \text{elist}_a + \text{ch}_a.\text{data} = \text{plist}) \\ & \wedge (\text{ch}_b.\text{sig} = \text{ch}_b.\text{ack} \ \rightarrow \text{clist} = \text{slist}_b) \\ & \wedge (\text{ch}_b.\text{sig} \neq \text{ch}_b.\text{ack} \ \rightarrow \text{clist} + \text{ch}_b.\text{data} = \text{slist}_b) \end{aligned}$$

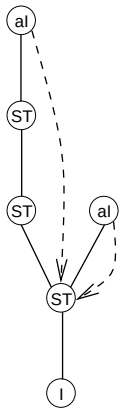
For the channel it is stated that all values that were written into the buffer are either still in the buffer or were already send to the consumer component. This is formalised with $\mathit{slist} + \mathit{chbuf} = \mathit{elist}_b$. Additionally, some prefix properties are needed to show the overall property:

$$\begin{aligned} I_2(V) :\Leftrightarrow \ & \text{clist} \sqsubseteq \text{slist}_b \wedge \text{slist}_b \ \sqsubseteq \ \text{slist}_a \wedge \text{slist}_a \ \sqsubseteq \text{elist}_b \\ & \wedge \text{elist}_a \ \sqsubseteq \ \text{elist}_a \wedge \text{elist}_a \ \sqsubseteq \text{plist} \end{aligned}$$

The overall invariant $I(V)$ is $I_1(V) \wedge I_2(V)$. The only needed initial information is, that the history-lists of both channel components are equal. This is formulated with $\text{init}_2 \equiv (\text{rlist}_a = \text{rlist}_b)$ and $\text{init}_3 \equiv (\text{slist}_a = \text{slist}_b)$.

All proof obligations were formally proven with KIV. To give an impression of the proof effort for the components, we describe as example proof of the temporal logic proof obligation i for channel_{rec}, which is as follows:

$$M_2, I(V), Init_2(V) \vdash A_2(V', V'') \xrightarrow{+} G_2(V, V')$$



The proof graph for this obligation is shown on the right side. In the beginning we start induction, explained in section 2.3. Initially the program is in position 1 (the numbers refer to the program of page 21). The *while*-loop could be evaluated, so that the program is in position 2. Executing the first step leads to a case distinction. Either the *await*-statement could be evaluated to *true* and the program is on position 3 or to *false* and the program remains at position 2. In the second branch induction is applied, as the sequent has not changed. In the first branch further steps are executed till the program is again at position 1, which has been encountered before. In this case induction is applied and the proof is finished. The other three temporal logic proof obligations can be verified analogously without additional effort.

The proofs for the predicate logic proof obligations are straight forward. They start with a case distinction of the conjunctions on the right side of the sequence. All premises can then be closed by the simplifier of KIV automatically.

All in all the reuse of the AGs is very high, for example every component that uses a handshake protocol has to fulfill the handshake guarantees. Only the invariant depends on the property we want to verify. All proofs are simple and can be automated to a large extend. One reason for this is, that the components are no longer interleaved after modularization and so symbolic execution leads to only few new cases.

5 Related Work and Summary

In summary, we have presented a method how to use symbolic execution together with compositional reasoning. As basis for our work we use an ITL variant [10] that supports symbolic execution. Furthermore it provides a compositional interleaving operator, which allows us to formulate an assumption-guarantee theorem and prove it on syntactic level. The logic is fully integrated into the interactive theorem prover KIV and all proofs where done within this tool. A further advantage of our logic is the possibility to directly include multiple system description languages into the logic formalism, e.g. SPL which is used in this work. Other languages that were also successfully integrated into the logic are Statemate

and UML statecharts [20, 21] as well as Asbru, a language used for the verification of medical protocols [22]. The tool support and the syntactic nature of the theorem simplifies adaption of the theorem to particularities of these languages (e.g. to have better support for events in statecharts). The ability of symbolic execution of programs and statecharts supports intuitive and understandable proofs. To our knowledge this is the first work combining symbolic execution with compositional reasoning.

Our compositional theorem is inspired by the work of Abadi and Lamport [17]. They introduced the $\overset{+}{\rightarrow}$ operator and a theorem which is suitable for safety and liveness properties. In comparison to our work they use conjunction for the composition of components. While conjunction is a more elementary operator than our interleaved operator, all components must be specified as stutter equivalent components. To achieve this, their components must be specified in a special formula in normal form, while we are able to specify the components directly in various description languages. Due to the inclusion of the double primed variables we have a stuttering mechanism directly in our semantics.

We use a similar technique for defining assumptions and guarantees as Cau and Collette [18]. Their theoretical work is more general as the described theorem can be adapted to state based as well as message based systems. Compared to this our focus was to provide a calculus and tool support for our technique.

Solanki et. al. [23] use compositional reasoning together with ITL. They use an AG variant that allows guarantees to be formulated in ITL. As tool they use (ana)Tempura [14, 11]. This technique is applied to a semantic web service description.

In a paper by Zwiers et. al. [24] invariants and preconditions are integrated in a compositional framework for concurrency. Joseph and Pandya [25] integrate invariants in a framework for total correctness. They use CSP-like distributed programs. Moszkowski [26] uses ITL for a compositional specification and proof technique. Further work about compositionality are e.g. Pnueli [27], Stirling [28] or Woodcock and Dickinson [29].

The producer-channel-consumer case study is a standard example for compositional reasoning. Pnueli [30] described a producer-channel-consumer example already 1986 formally with temporal logic. Abadi and Lamport [17] also used this example to illustrate how to specify components of concurrent systems. In their example they show that two N-element queues can be composed to an (2N+1)-element queue. Jonsson and Tsay [31] use the same example and property. The producer-channel-consumer example is also verified by Breitling et. al. [32], where streams for modelling the communicationare are used and Rock et. al. [33] in combination with TLA for specification.

Next steps are to apply our approach on liveness properties. First experiments in this direction were very promising. Another interesting topic would be to integrate an objectlevel $\overset{+}{\rightarrow}$ operator similar to [17]. This would allow us

to use more complex assumption guarantee properties without abandoning the advantages of our approach: symbolic execution and tool support with various system description languages.

References

1. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM **8**(9) (1965) 569
2. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4) (1983) 596–619
3. Misra, J., Chandi, K.: Proofs of networks of processes. IEEE Transactions of Software Engineering (1981)
4. de Roever, W.P., et al.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge University Press (2001)
5. Furia, C.A.: A compositional world: a survey of recent works on compositionality in formal methods. Technical Report 2005.22, Dipartimento di Elettronica e Informazione, Politecnico di Milano (March 2005)
6. Harel, D.: Dynamic logic. In Gabbay, D., Guenther, F., eds.: Handbook of Philosophical Logic. Volume 2. Reidel (1984) 496–604
7. Heisel, M., Reif, W., Stephan, W.: A Dynamic Logic for Program Verification. In Meyer, A., Taitslin, M., eds.: Logical Foundations of Computer Science. LNCS 363, Berlin, Logic at Botik, Pereslavl-Zalessky, Russia, Springer (1989) 134–145
8. Burstall, R.M.: Program proving as hand simulation with a little induction. Information processing 74 (1974) 309–312
9. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (1976) 385–394
10. Balser, M.: Verifying Concurrent Systems with Symbolic Execution. Shaker Verlag, Germany (2006)
11. Moszkowski, B.: Executing Temporal Logic Programs. Cambridge University Press, Cambridge (1986)
12. Balser, M., Reif, W., Schellhorn, G., Stenzel, K.: KIV 3.0 for Provably Correct Systems. In Hutter, D., Stephan, W., Traverso, P., Ullmann, M., eds.: Proc. Int. Wsh. Applied Formal Methods. Volume 1641 of LNCS., Springer (1999) 330–337
13. Balser, M., Reif, W.: Interactive verification of concurrent systems using symbolic execution. Technical Report 2008-12, Universität Augsburg (2008)
14. Cau, A., Moszkowski, B., Zedan, H.: ITL – Interval Temporal Logic. Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK. (2002) http://www.cse.dmu.ac.uk/STRL/ITL/.
15. Manna, Z., Pnueli, A.: Temporal verification diagrams. LNCS 789 (1994) 726–765 Springer-Verlag.
16. Balser, M., Reif, W.: An interval temporal logic with compositional interleaving. Technical Report 2008-11, Universität Augsburg (2008)
17. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems (1995)
18. Cau, A., Collette, P.: Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. Acta Inf. **33**(2) (1996) 153–176
19. Mead, C., Conway, L.: Introduction to VLSI systems. Addison-Wesley (1980)
20. Balser, M., Bäumler, S., Knapp, A., Reif, W., Thums, A.: Interactive verification of uml state machines. In Davies, J., Schulte, W., Barnett, M., eds.: Proc. 6th Int. Conf. of Formal Engineering Methods. Volume 3308 of LNCS., Springer (2004)
21. Thums, A.: Formale Fehlerbaumanalyse. PhD thesis, Universität Augsburg, Augsburg, Germany (2004) (in German).
22. Schmitt, J., Balser, M., Reif, W.: Asbru in KIV v2.1 – a tutorial. Technical Report 2006-03, University of Augsburg (2006)

23. Solanki, M., Cau, A., Zedan, H.: Augmenting semantic web service descriptions with compositional specification. In Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E., eds.: Proc. of 13th int. conference on World Wide Web, ACM (2004) 544–552
24. Zwiers, J., de Roever, W.P., van Emde Boas, P.: Compositionality and concurrent networks: Soundness and completeness of a proofsystem. In: Proc. of 12th Colloquium on Automata, Languages and Programming, Springer (1985) 509–519
25. Pandya, P.K., Joseph, M.: P-A logic: a compositional proof system for distributed programs. Distributed Computing **5**(1) (1991) 37–54
26. Moszkowski, B.: Compositional reasoning using interval temporal logic and tempura. LNCS 1536 (1996) 439–464 Springer-Verlag.
27. Pnueli, A.: In transition from global to modular temporal reasoning about programs. (1985) 123–144
28. Stirling, C.: A generalization of Owicki-Gries's Hoare logic for a concurrent while language. Theor. Comput. Sci. **58**(1-3) (1988) 347–359
29. Woodcock, J.C.P., Dickinson, B.: Using VDM with rely and guarantee-conditions. Experiences from real projects. In: Proceedings of the 2nd VDM-Europe Symposium on VDM—The Way Ahead, New York, NY, USA, Springer-Verlag New York, Inc. (1988) 434–458
30. Pnueli, A.: Applications of temporal logic to the specification and verification of concurrent systems: A survey of current trends. LNCS 224, Berlin, Springer (1986)
31. Jonsson, B., Tsay, Y.K.: Assumption/guarantee specifications in linear-time temporal logic. Theoretical Computer Science, Vol. 167 (1996)
32. Breitling, M., Philipps, J.: Black box views of state machines. Technical Report TUM-I9916, Technische Univerität München (1999)
33. Rock, G., Stephan, W., Wolpers, A.: Modular reasoning about structured tla specifications. In Berghammer, R., Lakhnech, Y., eds.: Tool Support for System Specification, Development and Verification, Springer (1999)

Specification Predicates with Explicit Dependency Information

Richard Bubel[1], Reiner Hähnle[1], and Peter H. Schmitt[2]

[1] Dept. of Computer Science and Engg., Chalmers Univ. of Technology
bubel|reiner@chalmers.se
[2] Dept. of Computer Science, Univ. of Karlsruhe
pschmitt@ira.uka.de

Abstract. Specifications of programs use auxiliary symbols to encapsulate concepts for a variety of reasons: readability, reusability, structuring and, in particular, for writing recursive definitions. The definition of these symbols often depends implicitly on the value of other locations such as fields that are not stated explicitly as arguments. These hidden dependencies make the verification process substantially more difficult. In this paper we develop a framework that makes dependency on locations explicit. This allows to define general simplification rules that avoid unfolding of predicate definitions in many cases. A number of non-trivial case studies show the usefulness of the concept.

1 Introduction

In program logics, especially in logics that target object-oriented languages, state-dependent predicates or functions are a convenient and often necessary concept used in specifications. They allow one to keep specifications concise and easy to read for humans. They are indispensable for the specification of inherently recursive properties such as reachability. Especially in first-order program logics there is no other alternative to specify properties recursively.

Such state-dependent predicate or function symbols, which are sometimes called *non-rigid symbols*, are not straightforward to use in verification practice, because they require special inference techniques. To unpack and transform their definition after every single state change would be extremely inefficient and must be avoided. As a first example, we consider the frequent specification task that stipulates an object array a to contain only non-null references. It is convenient to define a non-rigid unary predicate symbol on arrays:

$$\text{nonNullArray}(a) :\Leftrightarrow \forall i.a[i] \neq \text{null}$$

A typical desirable property in this context is that a simple assignment to a program variable `j` does not change the validity of nonNullArray. The formulation in Hoare logic [11] is in the first line below, the second line is the same in Dijkstra's weakest precondition calculus [9], and the third line reformulates it in Dynamic Logic [10]:

$$\{\text{nonNullArray}(a)\}\ \texttt{j} := \texttt{j} + \texttt{1};\ \{\text{nonNullArray}(a)\}$$
$$\text{nonNullArray}(a) \ \rightarrow\ wp(\texttt{j} := \texttt{j} + \texttt{1},\ \text{nonNullArray}(a))$$
$$\text{nonNullArray}(a) \ \rightarrow\ \langle \texttt{j} := \texttt{j} + \texttt{1};\rangle \text{nonNullArray}(a)\ .$$

To prove this claim naïvely entails unpacking the definition of nonNullArray or to define a special-purpose predicate transformer having specific knowledge about the state dependencies in its definition. For example, the definition of nonNullArray contains an implicit dependency on a and on $a[i]$ for all i, but not on any integer program variable `j`. The contribution of this paper is a technique that makes this kind of implicit dependency information explicit in the symbol's syntax. This leads to significantly higher automation. The presented approach has been evaluated in several small examples, e.g. verification of a selection sort algorithm, but also to verify an implementation of the Schorr-Waite algorithm. It allows to formulate *uniform* predicate transformers that can exploit generic dependencies and that are applicable in a variety of situations. This paper is partly based on results first published in the first co-author's dissertation [8].

The paper is structured as follows. The program logic used in this paper is introduced in Sect. 2. Syntax and semantics for the explicit dependency notation called *location descriptors* are presented in Sect. 3. In Sect. 4 we show how to prove that the chosen location descriptor is consistent with the predicate's definition. Predicate transformers in the form of simplification rules that take advantage of explicit dependencies are presented in Sect. 5. Sect. 6 presents applications and case studies that show the usefulness of the concept of location descriptors. Sect. 7 credits related work and outlines future work.

2 Dynamic Logic with Updates

This section sketches the program logic used throughout the paper. We use object-oriented dynamic logic (ODL) [4] which extends standard dynamic logic [10] to cover all essential features of object-orientation, but is small enough for theoretical purposes.

The programming language used in ODL is essentially a stripped down version of JAVA. It supports all concepts except for inner and anonymous classes, floating point arithmetic, threads (and, hence, GUI). Additionally, ODL does not support dynamic method binding or exceptions. Some of these restrictions stem from open scientific problems (floating points, threads), others are a consequence of the goal to define a minimalistic object-oriented language into which all aspects of realistic languages can be compiled without much overhead. We concentrate on the language aspects essential for the paper and leave out, for example, object allocation or exceptions. For a full account see [4]. When convenient we use the style of presentation given in [3] for the logic JAVA CARD DL.

Definition 1 (Signature). *Let $\mathcal{T}_{all} := \{\top, boolean, int, \bot\} \cup \mathcal{T}_d$ denote a finite set of types. $(\mathcal{T}_{all}, \sqsubseteq)$ forms a complete lattice with respect to the partial order $\sqsubseteq$ (modelling the subtype hierarchy). $\mathcal{T}_d$ is the set of all reference types (closed by intersection) containing the* `Null` *type as least element. Further, $\mathcal{T} := \mathcal{T}_{all} - \{\bot\}$ denotes the set of all types except the bottom type.*

$\Sigma_{\mathcal{T}} := \{Q, Op, Mod, U, \Pi, \mathrm{PSym}, \mathrm{FSym}, \mathrm{VSym}\}$ *is a typed signature over* $\mathcal{T}$*, where* Q, Op *are sets containing the classical first-order quantifiers and operators.* Mod *contains the box* $[\cdot]$ *and diamond* $\langle\cdot\rangle$ *modalities.* U *is the set of elementary updates (defined below) and* Π *the set of programs.* PSym, FSym *are typed predicate and function symbols with arity function* α *assigning each symbol its signature.* VSym *is a set of typed first-order logic variables.*

As mentioned in Sect. 1 we distinguish between rigid and state-dependent (non-rigid) function and predicate symbols with the following notation: $\mathrm{PSym} = \mathrm{PSym}_r \cup \mathrm{PSym}_{nr}$ and $\mathrm{FSym} = \mathrm{FSym}_r \cup \mathrm{FSym}_{nr}$. It is useful to single out from the non-rigid function symbols FSym_{nr} the subset of location function symbols, i.e., functions used to represent local program variables, fields and arrays that can be changed by a program or update:

Definition 2 (Location Function Symbols). *The set of* location function symbols $\mathrm{FSym}_{loc} \subseteq \mathrm{FSym}_{nr}$ *contains for each arity an infinite number of symbols including*

- *for each program variable pv used in an ODL program a constant symbol with the same name and type;*
- *for each attribute a of type T declared in a class C of the ODL program, a unary location function symbol* $a@(C) : C \to T$*;*
- *the array access operator* $[] : \top \times \mathit{int} \to \top$*.*

Note 1. ODL features separate syntactic categories for program and logic (first-order) variables. Program variables are modelled as non-rigid constants and cannot be quantified. Logic variables are rigid, quantifiable, and must not occur inside programs. We omit the $@(C)$ suffix from attribute names if no ambiguity arises and write $o.a$ instead of $a(o)$ for attribute lookup expressions.

Definition 3 (Updates). *Given a location function* f *and terms* o_i *and* v*. Then the expression* $f(o_1, \ldots, o_n) := v$ *denotes an* elementary update*. General updates are defined inductively. Let* U, U_1, U_2 *be updates, then all of the following expressions are updates as well:*

- *the sequential composition* $U_1 \,; U_2$*,*
- *the parallel composition* $U_1 \,||\, U_2$*,*
- *the conditional* $\backslash\texttt{if}\ (\phi)$*;* U *(where* ϕ *is a formula), and*
- *the quantification* $\backslash\texttt{for}\ x$*;* U *(binding the first-order variable* x *in update* U*).*

Definition 4 (Terms and Formulae). *The inductive definition of terms and formulae is as usual for typed first-order dynamic logic. We define only the less common cases:*

- *Let* U *be an update and* ξ *a term (formula), then* $\{U\}\,\xi$ *is a term (formula).*

– *Let ϕ be a formula and p a program, then $[p]\phi$ (partial correctness) and $\langle p\rangle\phi$ (total correctness) are formulae.*

The following definitions formalise the semantics of formulae and updates.

Definition 5 (ODL Kripke Structure). *An ODL Kripke structure $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ consists of*

– *A partial first-order model $\mathcal{M} = (\mathcal{D}, I_0)$ providing a domain mapping $\mathcal{D}$ that assigns to each type its domain (with $\mathcal{D}(int) = \mathbb{Z}$) and a partial interpretation I_0 for all rigid and non-rigid predicate symbols:*

$$I_0(q) := \begin{cases} \uparrow\ , & \text{if } q \in \mathrm{PSym}_{nr} \\ G\ , & \text{if } q \in \mathrm{PSym}_{r},\ \text{for some } G \subseteq \mathcal{D}(T_1) \times \ldots \times \mathcal{D}(T_n) \end{cases}$$

where $q : T_1 \times \ldots \times T_n$ is a predicate symbol (analogous for function symbols)
– *A set of states $\mathcal{S}$ where each $S \in \mathcal{S}$ contains an interpretation I_{nr} completing the partial interpretation I_0 to a total interpretation $I := I_0 \dot\cup I_{nr}$ by assigning a meaning to all non-rigid symbols.*
– *The state transition relation ρ defining the programs' semantics, where for a program p and two states $S_1, S_2 \in \mathcal{S}$ the relation $\rho(p)(S_1, S_2)$ holds if and only if executing p in state S_1 terminates in the final state S_2. As ODL programs are deterministic the final state is unique whenever it exists.*

A function $\beta : \mathrm{VSym} \to \bigcup_{T\in\mathcal{T}} \mathcal{D}(T)$ assigning to all logic variables an element of the universe of the appropriate type is a variable assignment*.*

Definition 6 (Semantic Location). *A* semantic location *is defined as a tuple $\langle f, (e_1, \ldots, e_n)\rangle$ where $f : T_1 \times \ldots \times T_n \to T$ is a location function symbol as in Def. 2 and e_i (for $i \in \{1, \ldots, n\}$) are elements in $\mathcal{D}(T_i)$.*

Definition 7 ((Consistent) Semantic Update). *An* elementary semantic update *is a pair $(\langle f, (e_1, \ldots, e_n)\rangle, d)$ where $\langle f, (e_1, \ldots, e_n)\rangle$ is a semantic location with $f : T_1 \times \ldots \times T_n \to T$ and d an element of $\mathcal{D}(T)$. A (possible empty) set of elementary semantic updates is called* semantic update*. A semantic update is called* consistent*, if it contains for any semantic location at most one* elementary semantic update*.*

Definition 8 (Application of a Consistent Semantic Update). *The application of a consistent semantic update CU is a mapping between states. Applying CU in a state S maps it to the state $S' = CU(S)$ that coincides on the value of all location function with S except for the semantic locations occurring in CU: whenever $(\langle f, (e_1, \ldots, e_n)\rangle, d) \in CU$ then $S'(f)(e_1, \ldots, e_n)$ evaluates to d.*

Updates are an explicit notation to capture symbolic state changes. The definition of a semantics for updates is rather technical due to clashes when

an update assigns different values to the same location. Sequential composition describes two successive state changes, while parallel composition updates the locations simultaneously and may cause clashes by updating the same location with differing values, e.g., $l := t_1 \,||\, l := t_2$. We use a last-one-wins clash resolution, i.e., in the resulting state location l has the value t_2. Quantified updates allow us to represent state changes of infinitely many locations. Resolution of clashes caused by quantified updates is left out for space reasons (see [3, 18]).

Example 1. Some updates and their intended semantics:

- The elementary update $\mathtt{x} := t$ assigns to the program variable $\mathtt{x}$ the value t. Applying it to a program variable $\mathtt{y}$ in $\{\mathtt{x} := t\}\ \mathtt{y}$ results in t if $\mathtt{x}$ and $\mathtt{y}$ are the same variable, otherwise, the term evaluates to $\mathtt{y}$.
- The parallel update $\mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{x}$ swaps the content of the program variables $\mathtt{x}$ and $\mathtt{y}$. Parallel updates are evaluated simultaneously and, therefore, are independent of each other. More formally, let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ be an ODL Kripke structure, $S \in \mathcal{S}$ and β a variable assignment. Evaluating the above update under $(\mathcal{K}, S, \beta)$ results in the consistent update CU mapping S to a state $CU(S)$ coinciding with S except for the program variables $\mathtt{x}$ and $\mathtt{y}$ whose values in $(\mathcal{K}, S, \beta)$ are swapped.
- The quantified update $\backslash\mathtt{for}\ i;\ a[i] := 0$ assigns 0 to all components of a.

We continue by defining the semantics of terms and formulae:

Definition 9 (Semantics of Terms and Formulae). *The inductive semantic definitions are as usual. We list only a few non-obvious cases, full definitions are in [3, 4].*

Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ denote an ODL Kripke structure, $t^{(\mathcal{K},S,\beta)}$ the evaluation of term t in state S under a variable assignment β, and $\models$ the validity relation.

- $(\{\mathcal{U}\}\ t)^{(\mathcal{K},S,\beta)} := t^{(\mathcal{K},S',\beta)}$ *where* $S' = \mathcal{U}^{(\mathcal{K},S,\beta)}(S)$
- $S, \beta \models \langle p \rangle \phi$ $(p \in \Pi)$ *iff a state* S' *exists with* $S', \beta \models \phi$ *and* $\rho(p)(S, S')$
- $S, \beta \models [p]\phi$ $(p \in \Pi)$ *iff* $S', \beta \models \phi$ *holds for any state* S' *with* $\rho(p)(S, S')$
- $S, \beta \models \{\mathcal{U}\}\ \phi$ *iff* $S', \beta \models \phi$, *where* $S' = \mathcal{U}^{(\mathcal{K},S,\beta)}(S)$

Later in the paper we need means to relate two arbitrary states on the syntactic level. For this we use a special kind of update called *anonymous program update* whose purpose is to perform a state transition to an unspecified state. Anonymous programs of this kind are well-known from propositional dynamic logic [10]. They are deterministic and terminating.

Definition 10 (Anonymous Program, Anonymous Program Update). *The atomic programs st_1, $st_2 \ldots$ are called (elementary)* anonymous programs. *The set of programs Π is extended accordingly. Further, we extend the inductive definition of updates by including the elementary* anonymous program update *ω_i for each anonymous program st_i.*

Definition 11 (Semantics of Anonymous Program Updates). *An anonymous program st_i is interpreted in an ODL Kripke Structure $\mathcal{K}$ such that for all $S \in \mathcal{S}$ there exists exactly one state $S' \in \mathcal{S}$ such that $\rho(st_i)(S, S')$ holds. An* anonymous program update *ω_i is then evaluated to a state transformer such that $\omega_i^{(\mathcal{K},S,\beta)}(S) = S'$ for all variable assignments β.*

3 Symbols with Explicit Dependencies

In this section we introduce a syntactic notation for non-rigid symbols that renders explicit the implicit dependencies on the state in their definition.

3.1 Location Descriptors

We need a notation to describe sets of locations. We use *location descriptors* introduced in the KeY-system [3] for modifies/assignable clauses. The origin of this notation goes back to quantified updates [18], see also Sect. 2. Location descriptors permit a compact and extensional characterisation of location sets.

Definition 12 (Location Descriptor). *A* location descriptor *has the form*

$$\texttt{\textbackslash for } x_1, \ldots, x_n; \texttt{\textbackslash if } (\Phi)\ loc$$

where (i) $x_1, \ldots, x_n$ are variables bound in Φ and loc, (ii) Φ is an arbitrary formula, and (iii) loc is a term with a location function symbol as top level operator, i.e., a program variable, an attribute function or the array access function. Except for $x_1, \ldots, x_n$ no other free *variables occur in Φ or loc.*

Location descriptors ld_1, ld_2 can be accumulated to sets of location descriptors by concatenation: $ld_{new} := ld_1 \ ; \ ld_2$. In case no variables are bound or Φ is identical to true, the corresponding parts in the syntax can be omitted.

Example 2. Here are some typical usages of location descriptors:

- $\texttt{\textbackslash for List } x;\ x.\texttt{next}$ capturing all `next` locations of `List`-typed elements. Note that the guard has been omitted here.
- $\texttt{\textbackslash for } T[]\ a, \text{int } i; \texttt{\textbackslash if } (i >= 0 \wedge i < a.\texttt{length})\ a[i]$ meaning all T-typed array component locations with indexes between 0 and the array length.[3]
- $\texttt{\textbackslash for Tree } t;\ t.\texttt{left} \ ; \texttt{\textbackslash for Tree } t;\ t.\texttt{right}$ capturing all `left` and `right` locations of `Tree`-typed elements.

Definition 13 (Location Descriptor Extension). *Let $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$ be an ODL Kripke structure with universe $\mathcal{D} = \bigcup_{T \in \mathcal{T}} \mathcal{D}(T)$. The* extension of a location descriptor $ld = \texttt{\textbackslash for } x_1, \ldots, x_n; \texttt{\textbackslash if } (\Phi)\ l(t_1, \ldots, t_n)$ *in a given state $S \in \mathcal{S}$ is defined as the set of semantic locations:*

$$ld^{(\mathcal{K},S)} \ := \ \{\langle\, l,\, (t_1, \ldots, t_n)^{(\mathcal{K},S,\beta)} \rangle \ \mid \ (\mathcal{K}, S, \beta) \models \Phi,\ \beta : \text{VSym} \to \mathcal{D}\}$$

Concatenation of location descriptors is evaluated as union of their extensions.

[3] The attribute `length` returns the length for each array and is unspecified otherwise.

Note 2. This definition works for any kind of location descriptor. In particular, the guard formula Φ and possible subterms of the location term can be arbitrary ODL formulae and terms that may contain non-rigid symbols or even programs.

3.2 Syntax and Semantics of Symbols with Explicit Dependencies

The notation introduced above provides a concise way to characterise sets of locations. Now we extend the names of non-rigid (predicate and function) symbols by qualifications in the form of location descriptors. The idea is that the value of thus qualified symbols depends at most on the values of their arguments *plus those locations contained in the extension of their location descriptor.*

Definition 14 (Symbols with Explicit Dependencies). *Let ld denote a semicolon-separated list of location descriptors and let $p : T_1 \times \cdots \times T_n$ be a non-rigid predicate. The non-rigid predicate symbol $p[ld] : T_1 \times \cdots \times T_n$ is called* predicate with explicit dependencies. *Analogously for functions.*

The definitions of terms and formulae remain unchanged. The only difference is that the signature contains the above defined location-dependent symbols.

There are only few restrictions on the interpretation of ordinary non-rigid function and predicate symbols: essentially, their interpretation has to be well-defined and respect their types. The situation is different for symbols with explicit dependencies. The interpretation of a symbol $p[ld]$ with explicit dependencies has to coincide on all states S_1, S_2 that share the same values for the locations described by ld. This is precisely formulated in the next two definitions.

Definition 15 (ld-Equivalence). *For any Kripke structure $\mathcal{K}$ we define for any location descriptor ld the equivalence relation $\approx_{ld}$ on states where $S_1 \approx_{ld} S_2$ iff the following two conditions hold:*

1. $ld^{(\mathcal{K},S_1)} = ld^{(\mathcal{K},S_2)}$ *(identical location descriptor extension) and*
2. $f^{(\mathcal{K},S_1)}(d_1, \ldots, d_n) = f^{(\mathcal{K},S_2)}(d_1, \ldots, d_n)$ *for any* $\langle f, (d_1, \ldots, d_n)\rangle \in ld^{(\mathcal{K},S_1)}$.

The additional restriction required for Kripke structures to accomodate symbols with explicit dependencies is now covered by the definition:

Definition 16 (Dependency-Consistent Kripke Structure). *We call an ODL Kripke structure $\mathcal{K} := (\mathcal{M}, \mathcal{S}, \rho)$* dependency-consistent *if for any predicate $k[ld]$ (function $f[ld]$) depending on ld and any two states S_1, S_2 with $S_1 \approx_{ld} S_2$ the evaluation of predicate $k[ld]$ (function $f[ld]$) in S_1 and S_2 coincides.*

Lemma 1. *Let $\mathcal{K}$ be a dependency-consistent Kripke structure. Then in any two states S_1, $S_2 \in \mathcal{S}$ with $t_i^{(\mathcal{K},S_1)} = t_i^{(\mathcal{K},S_2)}$ ($i \in \{1 \ldots n\}$) the atomic formula $p[ld](t_1, \ldots, t_n)$ evaluates to the same truth value whenever $S_1 \approx_{ld} S_2$ holds. Analogously for location dependent function symbols.*

Note 3. The notions of satisfiability, validity and model carry over to dependency-consistent Kripke structures in a straightforward manner. From now on we deal only with dependency-consistent Kripke structures and the corresponding notions of model, validity, etc., if not stated otherwise.

Example 3. Instead of modelling nonNullArray as a non-rigid predicate as done on p. 28, it can now be modelled as a predicate symbol with explicit dependencies: nonNullArray[\for $T[]$ a, int i; $a[i]] : T[]$. This expresses explicitly that its value depends only on the value of the components of $T[]$-typed arrays and, of course, on its argument.

4 Correct Definitional Extensions

Definitional extension in first-order logic preserves consistency, i.e., adding a new axiom of the form $\forall \overline{x}(p(\overline{x}) \leftrightarrow \phi)$ for a new predicate symbol p not occurring in ϕ will not introduce new inconsistencies. For predicates with explicit dependencies an axiom of the form $\forall \overline{x} : \overline{T_{p[ld]}};\ (p[ld](\overline{x}) \leftrightarrow \phi)$ may already by inconsistent with respect to the dependency semantics of Def. 16: one has to ensure that the implicit state dependencies of ϕ are reflected in the location descriptor ld. The problem only concerns the implication $\forall \overline{x} : \overline{T_{p[ld]}};\ (p[ld](\overline{x}) \rightarrow \phi)$. We will consider only *implicational* axioms of this type and, to simplify the presentation, we assume there is only one axiom for each defined predicate. These axioms will be exploited during proof search as rewrite rules $p[ld](t_1, \ldots, t_n) \rightsquigarrow \phi(t_1, \ldots, t_n)$ named $\mathsf{axiom}_{p[ld] \rightarrow \phi}$, where the t_i's are terms of the appropriate type. In this setting we allow the defined symbol to occur recursively on the right-hand side.

We intend to define a proof obligation that, when valid, ensures a given implicational axiomatisation of a location-dependent symbol to be consistent with respect to dependency semantics. The problem of termination in the case of recursive rewrite rules is a different matter and has to be dealt with separately.

We need a new concept called *anonymising update for location descriptors*: this is a quantified update for a given location descriptor that assigns all locations in its extension a fixed, but unknown value:

Definition 17 (Anonymising Update for Location Descriptors). *Let* $ld := \texttt{\textbackslash for}\ \overline{T}\ \overline{o}; \texttt{\textbackslash if}\ (\phi)\ f(\overline{t})$ *be a location descriptor. The quantified update*

$$\mathcal{V}_{ld} := \texttt{\textbackslash for}\ \overline{T}\ \overline{o};\ \texttt{\textbackslash if}\ (\phi);\ f(\overline{t}) := c(\overline{t})$$

with c *being a fresh uninterpreted rigid function symbol of matching type and arity is called an* anonymising update *for the location descriptor* ld*. For* $ld = ld_1; \ldots; ld_n$ *we define* $\mathcal{V}_{ld} = \mathcal{V}_{ld_1} \parallel \ldots \parallel \mathcal{V}_{ldn}$*.*

With the help of anonymising updates it is possible to formulate dependence consistency (Def. 16) directly as the proof obligation

$$po_{p[ld] \rightarrow \phi} \quad := \quad \forall \overline{x} : \overline{T_{p[ld]}};\ ((\{\omega_c\}\ \{\mathcal{V}_{ld}\}\ \phi(\overline{x})) \leftrightarrow \{\mathcal{V}_{ld}\}\ \phi(\overline{x}))) \tag{1}$$

where ω_c is a fresh anonymous program update (Def. 10).

Theorem 1. *If* (1) *is logically valid, then* $\forall \overline{x} : \overline{T_{p[ld]}}; (p[ld](\overline{x}) \to \phi)$ *is consistent with respect to dependency semantics.*

5 Simplification Rules

In this section we introduce two update simplification rules that can be applied to arbitrary atomic formulas with explicit dependencies. We make use of the following Lemma [3, Sect. 3.9], [18]:

Lemma 2. *Two updates* $\mathcal{U}_1$, $\mathcal{U}_2$ *are called* equivalent *if they induce the same state transformer in any Kripke structure. Then every update* $\mathcal{U}$ *that has no anonymous program update as a component is equivalent to an update of the form* $upPart_1 \,||\, \ldots \,||\, upPart_{len}$ *with* $upPart_i := \texttt{\textbackslash for}\ \overline{T_i}\ \overline{x_i}; \texttt{\textbackslash if}\ (\tau_i); g_i(\overline{u_i}) := val_i$.

Crucial for the first simplification rule is the notion of *relevant location symbol*. Relevant location symbols are a syntactic approximation of the location symbols that a formula ϕ or a term t may depend on. The notation is $Loc(\phi)$ and $Loc(t)$. In the definition below, and later, we use the following notation:

- $h[ld](s_1, \ldots, s_n)$ stands for a function or predicate symbol with explicit dependencies, $ld = ld_1; \ldots; ld_q$;
- $ld_i := \texttt{\textbackslash for}\ T_{i1}\ o_{i1}; \ldots; T_{ir_i}\ o_{ir_i}; \texttt{\textbackslash if}\ (\phi_i)\ f_i(t_{i1}, \ldots, t_{i\alpha_i})$

For ease of presentation we assume that $f_i \neq f_j$ for $i \neq j$. This affects the formula $\psi_{\mathcal{U}_1,\mathcal{U}_2,ld}$ in Def. 20. It is not hard to see how to adapt this formula to the unrestricted case.

Definition 18 (Relevant Location Symbols).

$Loc(\neg\psi)$	$:= Loc(\psi)$	
$Loc(\mathcal{Q}\ T\ x; \psi)$	$:= Loc(\psi)$	$\mathcal{Q} \in \{\forall, \exists\}$
$Loc(\psi \circ \phi)$	$:= Loc(\psi) \cup Loc(\phi)$,	$\circ \in \{\wedge, \vee, \to, \ldots\}$
$Loc(h(s_1, \ldots, s_n))$	$:= \bigcup_{i=1}^{n} Loc(s_i)$,	*h is a rigid symbol*
$Loc(\langle prg \rangle \psi)$	$:= \mathrm{FSym}_{loc}$	*see Def. 2*
$Loc([prg]\psi)$	$:= \mathrm{FSym}_{loc}$	*see Def. 2*
$Loc(g(s_1, \ldots, s_n))$	$:= \{g\} \cup \bigcup_{i=1}^{n} Loc(s_i)$	*g is a location fct. symbol*
$Loc(ld)$	$:= \bigcup_{i=1}^{q} \big(Loc(f_i(t_{i1}, \ldots, t_{i\alpha_i})) \cup Loc(\phi_i)\big)$	*ld as stipulated above*
$Loc(h[ld](s_1, \ldots, s_n))$	$:= Loc(ld) \cup \bigcup_{i=1}^{n} Loc(s_i)$	
$Loc(h(s_1, \ldots, s_n))$	$:= \mathrm{FSym}_{loc}$	*if h is a non-rigid symbol, but not a location symbol and without explicit dep.*
$Loc(\mathcal{U})$	$:= \bigcup_{i=1}^{len} (Loc(g_i(\overline{u_i})) \cup Loc(\tau_i) \cup Loc(val_i))$	$\mathcal{U}$ *as in Lemma 2*
$Loc(\omega_c)$	$:= \mathrm{FSym}_{loc}$	*see Defs. 2, 10*
$Loc(\{\mathcal{U}\}\xi)$	$:= Loc(\mathcal{U}) \cup Loc(\xi)$	ξ *a formula or term*

Lemma 3. *Let ϕ be an arbitrary formula, $\mathcal{K}$ a Kripke structure, S a state in $\mathcal{K}$, $\mathcal{U}$ as in Lemma 2 with $g_i \notin Loc(\phi)$ for all i then*

$$(\mathcal{K}, S) \models \phi \textit{ iff } (\mathcal{K}, S) \models \{\mathcal{U}\}\phi$$

This lemma guarantees the correctness of the following rule.

Definition 19 (Coincidence Simplification Rule).

$$\{\mathcal{U}\}\ k[ld](s_1, \ldots, s_n) \rightsquigarrow \{\mathcal{U}'\}\ k[ld](s_1, \ldots, s_n)$$

where $\mathcal{U}' := \mathcal{U} - \{upPart_i \mid g_i \notin Loc(k[ld](s_1, \ldots, s_n))\}$ *and* $\mathcal{U}$ *as in Lemma 2.*

Example 4. Here is an instance of the coincidence simplification rule, where s is a term with $\mathtt{j} \notin Loc(s)$: $(\{\mathtt{i} := iVal \,||\, \mathtt{j} := jVal\}\ p[\mathtt{i}](s)) \rightsquigarrow (\{\mathtt{i} := iVal\}\ p[\mathtt{i}](s))$

Example 5. The coincidence simplification rule allows to prove the motivating example on p. 28 easily: in $\text{nonNullArray}(a) \rightarrow \{\mathtt{j} := \mathtt{j} + 1\}\ \text{nonNullArray}(a)$ we use $\text{nonNullArray}[\backslash\mathtt{for}\ T[]\ a, \text{int}\ i;\ a[i]]$ as introduced in Example 3. For $\mathtt{j} \notin Loc(a)$ an instance of this rule simplifies

$$\{\mathtt{j} := \mathtt{j} + 1\}\ \text{nonNullArray}[\ldots](a) \rightsquigarrow \text{nonNullArray}[\ldots](a)$$

rendering the implication trivially valid.

The coincidence simplification rule is of an approximate nature but is sufficient for many practical purposes. We provide a stronger, semantic simplification rule in the form of the *equivalence simplification rule*. We require sufficient and necessary conditions for the logical equivalence of the two formulae $\{\mathcal{U}_1\}\ k[ld](s_1, \ldots, s_n)$ and $\{\mathcal{U}_2\}\ k[ld](s_1, \ldots, s_n)$. First we want the argument terms of $k[ld]$ to evaluate to the same values after the respective updates, i.e., $\{\mathcal{U}_1\}s_l \doteq \{\mathcal{U}_2\}s_l$ for all $1 \leq l \leq n$. Next we want to formalise that the locations described by each ld_j after update $\mathcal{U}_1$ are the same as those described by ld_j after update $\mathcal{U}_2$ and that their values coincide:

$$\begin{aligned}
&\psi_j^1 = \forall d; \forall \overline{c_j}; \forall \overline{o_j};\\
&\quad \big\{(\{\mathcal{U}_1\}\ (\phi_j \wedge \overline{c_j} \doteq \overline{t_j} \wedge d \doteq \mathtt{f}_j(\overline{t_j}))) \rightarrow \exists \overline{o_j}(\{\mathcal{U}_2\}\ (\phi_j \wedge \overline{c_j} \doteq \overline{t_j} \wedge d \doteq \mathtt{f}_j(\overline{t_j})))\big\}\\
&\psi_j^2 = \forall d; \forall \overline{c_j}; \forall \overline{o_j};\\
&\quad \big\{(\{\mathcal{U}_2\}\ (\phi_j \wedge \overline{c_j} \doteq \overline{t_j} \wedge d \doteq \mathtt{f}_j(\overline{t_j}))) \rightarrow \exists \overline{o_j}(\{\mathcal{U}_1\}\ (\phi_j \wedge \overline{c_j} \doteq \overline{t_j} \wedge d \doteq \mathtt{f}_j(\overline{t_j})))\big\}
\end{aligned}$$

It is now not hard to see that $\psi_{\mathcal{U}_1,\mathcal{U}_2,ld} = \bigwedge_{j=1}^{q}(\psi_j^1 \wedge \psi_j^2) \wedge \bigwedge_{l=1}^{n}\{\mathcal{U}_1\}s_l \doteq \{\mathcal{U}_2\}s_l$ is valid if and only if $\{\mathcal{U}_1\}\ k[ld](s_1, \ldots, s_n) \leftrightarrow \{\mathcal{U}_2\}\ k[ld](s_1, \ldots, s_n)$ is valid. This justifies the following rule:

Definition 20 (Equivalence Simplification Rule).

$$\{\mathcal{U}_1\}\ k[ld](s_1, \ldots, s_n) \rightsquigarrow \{\mathcal{U}_2\}\ k[ld](s_1, \ldots, s_n) \textit{ provided formula } \psi_{\mathcal{U}_1,\mathcal{U}_2,ld} \textit{ holds.}$$

Example 6. We modify Ex. 5 by using the more specialised 0-ary predicate $\text{nonNullArray}_a[\backslash\texttt{for}\ \text{int}\ i;\ a[i]]$ restricting the non-null element property to only those arrays referenced by the program variable a. Let

$$a \neq b \rightarrow (\text{nonNullArray}_a[\ldots] \rightarrow \{b[0] := \text{null}\}\ \text{nonNullArray}_a[\ldots])$$

the formula to be proven valid where b is an array of the same type as a. Note that we cannot apply the coincidence simplification rule here: $Loc(\backslash\texttt{for}\ \text{int}\ i;\ a[i])$ yields $[\cdot]$, i.e., the non-rigid array lookup function which is also the leading function symbol of the update $\{b[0] := \text{null}\}$. Now we apply the equivalence simplification rule which gives for formula ψ_1^1:

$$\begin{array}{l}\forall d{:}T; c_1 : T[]; c_2{:}\text{int}; i{:}\text{int};\\ \quad (\{b[0] := \text{null}\}\ (\mathit{true} \wedge c_1 \doteq a \wedge c_2 \doteq i \wedge d \doteq a[i])) \rightarrow\\ \qquad \exists i{:}\text{int};\ (\mathit{true} \wedge c_1 \doteq a \wedge c_2 \doteq i \wedge d \doteq a[i]))\end{array}$$

Under the assumption $a \neq b$ this can be easily proven with a first-order theorem prover after applying the update.

Note 4. The calculus with the equivalence simplification rule added is complete for the logic containing symbols with explicit dependencies relative to the calculus without those symbols.

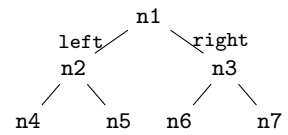
6 Applications and Case Studies

In this section we show applications of symbols with explicit dependency information modelling specification predicates such as reachability. Using these symbols provides advantages for interactive and automated proving. By being able to delay expanding the definition of a non-recursive predicate, the proof remains readable for a human reader. The automation benefits from the availability of general simplification rules reducing the need to look into the definition of non-rigid symbols. This is briefly illustrated in two case studies.

6.1 Specification predicates

Reachability. Treatment of linked data structures in specification and verification of object-oriented programs requires routinely to express reachability between two objects via a finite chain of fields. Fig. 1 shows a simple binary, directed tree, where subtrees are accessible via the fields `left` and `right`.

Specifying properties such as an element occurring in a given list requires to introduce and formalise the concept of reachability for these data structures. In the following we concentrate on the definition of a predicate formalising reachability for the directed, binary tree structure in Fig. 1. We aim to define a



Subtree at `n5` is reachable from the root `n1` via fields `left` and `right` in exactly 2 steps:
`reach[\for Tree` t; t`.left;\for Tree` t; t`.right](n1, n5, 2)`

Fig. 1. A binary tree datastructure with its associated reachable predicate symbol.

predicate that takes three arguments x, y (both of type `Tree`) and n (of type `int`) and holds if the subtree y can be reached from tree x in exactly n steps using only the fields `left` and `right`. The recursive definition for the reachability predicate is rather straightforward:

$$\texttt{reach}(x, y, n) :\Leftrightarrow \begin{cases} \text{false} & , \text{if } n < 0 \\ x \doteq y & , \text{if } n \doteq 0 \\ (x\texttt{.left} \not\doteq \text{null} \wedge \texttt{reach}(x\texttt{.left}, y, n-1)) \vee & \\ (x\texttt{.right} \not\doteq \text{null} \wedge \texttt{reach}(x\texttt{.right}, y, n-1)) & , \text{if } n > 0 \end{cases}$$

A calculus rule can be easily derived from the definition. The predicate is clearly non-rigid as it depends not only on the value of its arguments but also on the `left` and `right` fields of the subtrees below and including x. Therefore, we easily identify a location descriptor capturing all location dependencies, namely `\for Tree` t; t`.left`; `\for Tree` t; t`.right` describing the set of all `left`, `right` locations of all trees. Hence, the location-dependent predicate symbol for capturing reachability in our notation is:

$$\texttt{reach}[\texttt{\textbackslash for Tree}\ t;\ t\texttt{.left}; \texttt{\textbackslash for Tree}\ t;\ t\texttt{.right}](\texttt{Tree}, \texttt{Tree}, \text{int}) \ .$$

6.2 Selection Sort

Verification of a sorting algorithm consists of showing that the result is indeed sorted and that the result contains exactly the elements of the original collection, in other words, that the result is a permutation of the input. For the verification of a standard selection sort implementation in JAVA a specification predicate with explicit dependencies has been used to formalise permutation of two arrays.[4]

The permutation predicate `perm`[`\for` int[] o; int i; $o[i]$](int[], int[]) is defined with the help of a recursive count of the occurrences of all elements in both arrays. The predicate occurs once in the post condition stating that the resulting array is indeed a permutation of the original, and the second time in the loop invariant of the sorting algorithm.

The selection sort algorithm starts at the first array element and swaps it with the first minimal element encountered in the subsequent array continuing until

[4] This case study has been joint work and is also reported in [19], where the focus was an improved loop invariant rule.

the last element is reached. Therefore, it utilises two nested loops. To prove that both loops preserve the permutation property without predicates having explicit location dependencies requires several inductive subproofs. To substantiate this claim we look at the following formula[5] that occurs as subgoal during the proof and has to be proven valid:

$$\begin{aligned}&\{\texttt{\textbackslash for int } i;\ aCopy[i] := a0[i] \,||\, \texttt{\textbackslash for int } i;\ a0[i] := get0(a,i)\}\ \texttt{perm}[\ldots](a0, aCopy) \rightarrow\\ &\quad \{b10 := FALSE \,||\, b4 := TRUE \,||\, b5 := FALSE \,||\, \ldots \,||\\ &\quad\ \texttt{\textbackslash for int } i;\ aCopy[i] := a0[i] \,||\, \texttt{\textbackslash for int } i;\ a0[i] := get0(a,i) \,||\\ &\quad\ a0[idx1] := get0(a0, idx2) \,||\, a0[idx2] := get0(a0, idx1)\}\ \texttt{perm}[\ldots](a0, aCopy)\end{aligned}$$

The update of the formula in the implication's premise states a transition to a state where array $aCopy$ is a shallow copy of the *old* content of array $a0$. *Simultaneously* new values are assigned to the components of array $a0$. The complete formula in the premise states then that in this state $aCopy$ and $a0$ are permutations.

The update of the formula in the conclusion starts with a number of updates stemming from branch predicates irrelevant for the permutation property. The quantified updates assign the components of the arrays $aCopy$ and $a0$ the exact same values as it is the case for the premise formula. *But* the following two updates lead to a slightly different state with $a0[idx1]$ and $a0[idx2]$ swapped.

Obviously, the implication is valid, because a standard transposition lemma can be used to show preservation of the permutation property. The technical difficulty is that a straightforward application of this transposition lemma is not possible. One has to prove that all updates preceding the array locations including the twelve updates abbreviated by "..." have no effect on the permutation property. This is done inductively using a strengthening of the claim where for any possible value of $b10$, $b4$, $b5, \ldots$ the permutation property remains unchanged.

In contrast, the approach presented in this paper exploiting dependency information allows to prove the formula valid without induction. Application of the *Coincidence Simplification Rule* (Def. 19) simplifies the state representation in the conclusion so far that the application of the transposition lemma is directly possible. We were able to prove the permutation property for an executable JAVA implementation of selection sort with only one user interaction exploiting the fact that swapping exactly two elements in an array preserves the permutation property.

6.3 Schorr-Waite Algorithm

The most complex case study so far where predicates with explicit dependencies were used is the verification of a fully functional JAVA implementation of the

[5] Slightly simplified and beautified.

Schorr-Waite graph marking algorithm [20] for arbitrary finite graph structures. The abstract algorithm has been verified in several case studies [7, 6, 21, 16, 1, 5, 12], mostly for the case of binary graphs. Our case study, first reported in [8], is to our knowledge the first time that an executable JAVA implementation for the general case was verified.

The Schorr-Waite graph marking algorithm saves memory by avoiding to encode the taken path in the method call stack. Instead a small number of auxiliary variables and subtle pointer rotations are used to encode the backtracking path. Besides showing that all reachable nodes are visited, the challenge is to show that afterwards the graph structure is restored, because the traversal uses destructive pointer manipulations.

Specification predicates with explicit dependencies were employed to express reachability of two nodes in the graph structure and for a specification predicate that characterises the backtracking path. The reachability predicate has been specified similar to the one in Sect. 6.1. The predicate characterising the backtracking path is specified in such a way that it evaluates to true if a given node is an element of the currently taken path. This is clearly state-dependent as the backtracking path changes during execution.

7 Related and Future Work

The Java Modelling Language (JML) [14] supports a depends clause used to express on which other fields a model field depends. The depends clause is then used to extend assignable clauses appropriately in case they contain model fields. These depends clauses have been introduced by Leino [15], the main idea being to replace the occurrence of an abstract/model field a with a function symbol $a'(f_1, \ldots, f_n)$ that explicitly enumerates the fields it depends on.

Separation logic [17] requires to specify exactly those locations of the heap (alternatively, to separate a heap into two orthogonal heaps) that are necessary to prove a property. The local judgements are then generalised by application of a frame rule. This approach is in the following sense complementary to ours: in separation logic, the shape of the heap is made explicit and dependencies are lifted with the help of a frame axiom. We make location dependencies explicit and can, therefore, abstract away from the concrete layout of the heap. This seems more appropriate for target languages such as JAVA. Another advantage of our approach is that only standard typed first-order logic is used.

Dynamic frames as introduced in [13] provide a uniform treatment of modified locations and dependencies, which are separate concerns in the presented work allowing for more precision. Of particular interest is the preservance operator Ξf for a dynamic frame (set of locations) f expressing that a computation does depend on or modify it. This property can be translated to our framework

and logic. The other operators for dynamic frames deal with variants of modifies clauses in presence of object creation.

In [2] read effects are used for treating invocations of pure methods. Our approach is directly targeted at the level of specification predicates without the need to define them as pure boolean methods, it provides a compact and precise notion for the dependencies and works without an explicit heap presentation. Another concern of [2] is well-definedness of specifications containing pure methods. For our approach well-definedness is desirable, but will not cause unsoundness of the verification system.

Further related work focusses on data groups or similar concepts to support information hiding and encapsulation. This is ongoing work in the context of the KeY project.

Future work includes to exploit further application scenarios for predicates with explicit dependencies. Rümmer suggested to use them to achieve second-order like specification capabilities in a first-order dynamic logic through parameterisation with functions. This allows, for example, to use the same specification for the sum over all elements of an array $\sum_i f(\mathtt{a}[\mathtt{i}])$ for any function f. Further, we will investigate how to improve automation of the equivalence update simplification rule, i.e., in finding a suitable equivalent update $\mathcal{U}_2$.

8 Conclusion

We introduced non-rigid specification predicate (and function) symbols that explicitly list the set of locations their value may depend on. Then we presented a verification framework for such symbols with explicit dependencies. This framework consists of two parts: on the one hand a uniformly generated proof obligation that ensures correctness of a predicate with explicit dependencies with respect to its axiomatisation; on the other hand a number of general simplification rules that allow one to exploit explicit dependencies within the context of a proof. Several application scenarios common in program verification as well as two case studies support the usefulness of our approach. The implementation and case studies were done within the KeY-system [3], however, as pointed out in the introduction the problem of location-dependent predicates as well as the solution presented here can be transferred to other program logics based on weakest precondition reasoning.

Acknowledgments

We thank Philipp Rümmer and Wolfgang Ahrendt for valuable comments. Special thanks go also to Benjamin Weiß for valuable comments and in particular for pointing out a problem in the original version of Def. 8. We thank the anonymous referees for their valuable comments.

References

1. J. Abrial. Event based sequential program development: Application to constructing a pointer program. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proc. Formal Methods*, volume 2805 of *LNCS*, pages 51–74. Springer, September 2003.
2. Ádám Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.
3. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2006.
4. B. Beckert and A. Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proc. Intl. Joint Conference on Automated Reasoning, Seattle, USA*, volume 4130 of *LNCS*, pages 266–280. Springer-Verlag, 2006.
5. L. Birkedal, N. Torp-Smith, and J. Reynolds. Local reasoning about a copying garbage collector. In *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, Jan. 2004.
6. R. Bornat. Proving pointer programs in Hoare logic. In R. C. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 102–126. Springer, 2000.
7. M. Broy and P. Pepper. Combining algebraic and algorithmic reasoning: An approach to the Schorr-Waite algorithm. *ACM Trans. Program. Lang. Syst.*, 4(3):362–381, 1982.
8. R. Bubel. *Formal Verification of Recursive Predicates*. PhD thesis, Fakultät für Informatik, Univ. Karlsruhe, June 2007.
9. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
10. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, Oct. 2000.
11. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, Oct. 1969.
12. T. Hubert and C. Marché. A case study of C source code verification: the Schorr-Waite algorithm. In B. K. Aichernig and B. Beckert, editors, *Proc. 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 190–199. IEEE Press, 2005.
13. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006.
14. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, Feb. 2007. Draft revision 1.200.
15. K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, Caltech, 1995.
16. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
18. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In M. Hermann and A. Voronkov, editors, *Proc. Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia*, volume 4246 of *LNCS*, pages 422–436. Springer-Verlag, 2006.
19. S. Schlager. *Symbolic Execution as a Framework for Deductive Verification of Object-Oriented Programs*. PhD thesis, Fakultät für Informatik, Univ. Karlsruhe, Feb. 2007.
20. H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.
21. H. Yang. Verification of the schorr-waite graph marking algorithm by refinement, 2003. Unpublished, http://ropas.kaist.ac.kr/ hyang/paper/dagstuhl-SW.pdf.

Bitfields and Tagged Unions in C – Verification through Automatic Generation

David Cock

Sydney Research Lab., NICTA*, Australia

Abstract. We present a tool for automatic generation of packed bitfields and tagged unions for systems-level C, along with automatic, machine checked refinement proofs in Isabelle/HOL. Our approach provides greater predictability than compiler-specific bitfield implementations, and provides a basis for formal reasoning about these typically non-type-safe operations. The tool is used in the implementation of the seL4 microkernel, and hence also in the lowest-level refinement step of the L4.verified project which aims to prove the functional correctness of seL4. Within seL4, it has eliminated the need for unions entirely.

1 Introduction

In this paper we present a tool that automatically generates inline-able C functions to implement tagged unions of packed bitfield types, based on a simple domain-specific-language specification. We then generalise, and suggest a technique to exploit the desires of systems programmers to ease program verification.

The motivation for this work was the C implementation of the seL4 microkernel, and the needs of the associated L4.verified project [8, 3]. The seL4 microkernel [5, 7] is an evolution of the L4 family [17] for secure, embedded devices. The L4.verified project aims to prove its functional correctness. The need to produce code that can be verified with reasonable effort requires the disciplined use of 'ugly' programming idioms, those which violate the basic abstractions of the underlying semantic model. In our case, these are heap type aliases, i.e. unions, non-type-safe pointer accesses, and sub-machine-word manipulations. These violations occur commonly together, in the tagged union and bitfield construct. See Fig. 1 for an example from the OKL4 kernel [19], a current commercial implementation of L4. This pattern is very regular, and an obvious target for automation. Generation of this code is desirable for two reasons: First, via controlled tagged unions, it adds functionality to C in a disciplined, type-safe way. Second, bitfield implementations vary widely, both in performance and in actual behaviour between compilers, and even different versions of the same compiler. As a result, they are usually mistrusted by kernel programmers. In contrast, our generated code is fast, predictable and formally correct.

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

```
union {
    struct {
        BITFIELD2(word_t,
                type    : 2,
                tcb_p   : BITS_WORD - 2
        );
    } x;
    word_t raw;
};
```

Fig. 1. Example of combined union/bitfield usage. From OKL4 2.1, "include/caps.h", lines 92-100.

Our approach is to provide an opaque, abstract type, implementing the tagged-union/bitfield semantics, together with generated accessor functions. The tool automatically provides the proofs that the functions behave as expected. Whilst this is not a radically new idea, our approach is successful precisely because we target the regular low-level functions, which nonetheless comprise 14% of the code within seL4. This tool also provides a case study for the use of the C semantics of Tuch et al. [26], and Dawson's Isabelle/HOL library for machine words [4]. The remainder of the paper is laid out as follows: Section 2 introduces the specification language used to describe the bit-level layout of structures, Section 3 shows the C code generation framework, and Section 4 explains the framework of automatically generated proofs to allow reasoning without descending to the level of pointers and bit manipulation.

2 Specification Language

The tagged-union/packed-bitfield structure is useful in a number of contexts e.g. hardware-dictated page-table layouts, hardware register mapping, and highly optimised data structure storage. As a running example, we will consider a subset of the seL4/ARM capability representation. Capabilities are used as a proxy for authority, and we consider only two capability types: Null caps (`null_cap`) which function as placeholders, and Untyped caps (`untyped_cap`) which convey authority over a power of two sized block of memory. The capability is represented as a two word (64 bit) bitfield: Null caps contain no data other than the type tag, whereas Untyped caps have two fields: `capBlockSize` and `capPtr`, a pointer aligned on a 16-byte boundary.

The cap representations are specified as follows (the full grammar is included in Fig. 5). First the machine word size (32 bits for ARM) is specified with the `base` keyword:

```
base 32
```

Next, bitfield blocks are specified. Fields are listed from most-significant to least-significant bit (Fig. 2). The `padding` keyword introduces anonymous

padding space, to achieve the desired alignment, and the `field` keyword reserves space for a named field. In the `null_cap` example, `padding 32` reserves one empty machine word, `field capType 4` allocates a 4 bit field at the top of the second word, and `padding 28` explicitly fills the remainder of the second word. The trailing padding is mandatory where the fields do not fill the lower bits of the last word.

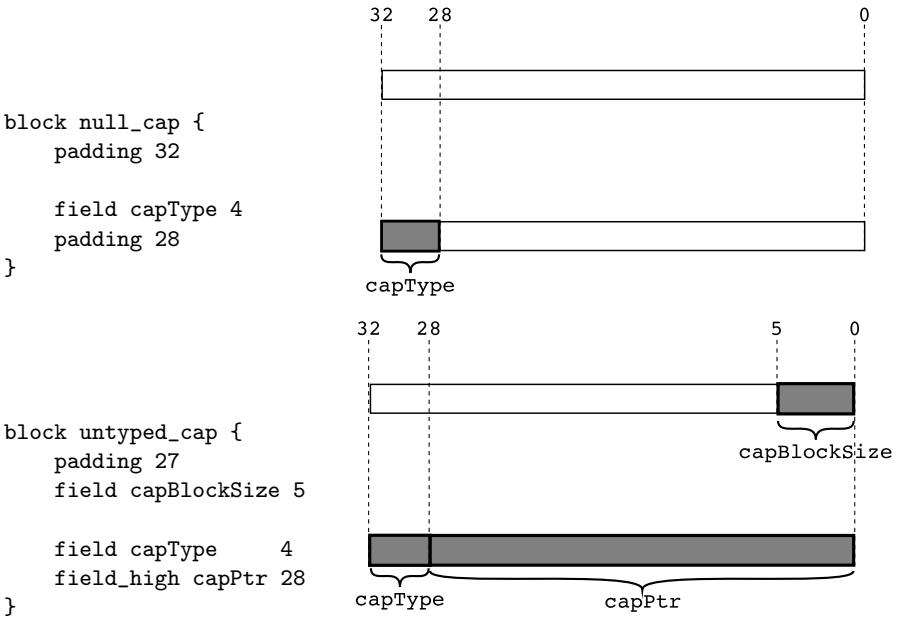


Fig. 2. Packed bitfield layout

Padding between fields is inserted explicitly, fields are forbidden to cross word boundaries, and the size of each block must be a multiple of the base word size. These restrictions ensure that the implementation maps efficiently onto common machine operations, and present no difficulties in practice. The `field_high` keyword specifies that a field should be left-aligned to the word size when read or written, padded on the right with zero bits, see Fig. 3.

Fig. 3. `field_high` implementation

```
tagged_union cap capType {
    tag null_cap 0
    tag untyped_cap 1
}
```

Fig. 4. Tagged union specification

Finally, blocks are grouped together into tagged unions (Fig. 4). The keyword `tagged_union` is followed by the name of the union, and the name of the tag field, then a list of block names, together with their associated tag values. All blocks in the union must be the same size, and each must contain a tag field. All tag fields must be the same size, and at the same location within the block.

```
entity_list ::= empty
              | entity_list block
              | entity_list tagged_union
              | entity_list base

base ::= "base" INTLIT

block ::= "block" IDENTIFIER "{" fields "}"

fields ::= empty
         | fields "padding" INTLIT
         | fields "field_high" IDENTIFIER INTLIT
         | fields "field" IDENTIFIER INTLIT

tagged_union ::= "tagged_union" IDENTIFIER IDENTIFIER "{" tags "}"

tags ::= empty
       | tags "tag" IDENTIFIER INTLIT
```

Fig. 5. Specification language grammar

3 Generated Code

This section gives a brief overview of the code generated from the specifications above. Each block and union in the specification language is translated to a C representation with appropriate access and update functions. Each object is represented by a struct containing simply an array of machine words, and for each union, an enum of tag values. The wrapping struct allows pass and return by value in C.

```
struct cap {
    uint32_t words[2];
};
typedef struct cap cap_t;
```

```
enum cap_tag {
    cap_null_cap = 0,
    cap_untyped_cap = 1,
};
typedef enum cap_tag cap_tag_t;
```

For each block and union, the tool generates create, access and update functions. Each such function is generated in a purely functional version, which passes and returns a stack object of appropriate struct type:

```
static inline cap_t CONST
cap_untyped_cap_set_capBlockSize(cap_t cap, uint32_t v) {
    assert(((cap.words[0] >> 28) & 0xf) ==
           cap_untyped_cap);

    cap.words[1] &= ~0x1f;
    cap.words[1] |= (v << 0) & 0x1f;
    return cap;
}
```

Also generated is a pointer lifted version, which operates indirectly on heap values through a supplied pointer:

```
static inline void
cap_untyped_cap_ptr_set_capBlockSize(cap_t *cap_ptr,
                                     uint32_t v) {
    assert(((cap_ptr->words[0] >> 28) & 0xf) ==
           cap_untyped_cap);

    cap_ptr->words[1] &= ~0x1f;
    cap_ptr->words[1] |= (v << 0) & 0x1f;
}
```

The prototypes for the remaining functions are given in Fig. 6.

Note that the generated API only provides functions that read the tag field through the union type, and no function to write it directly. This imposes a class-like behaviour on the types. The subtype is set implicitly at creation time, and can only be modified by overwriting with an object of a different type. This will turn out to be an important property for verification.

In practice the output is automatically pruned, so that only those functions actually used in the source are generated. This speeds the proof process. As the specification language is highly focussed and carefully limited, the generated code is simple and fast, highly predictable, and easily inlined by the compiler.

4 Generated Specifications

The final and most novel part of the approach consists of the automatically generated, machine checked function specifications together with their automated

```
static inline uint32_t CONST
cap_get_capType(cap_t cap);

static inline uint32_t PURE
cap_ptr_get_capType(cap_t *cap_ptr);

static inline cap_t CONST
cap_untyped_cap_new(uint32_t capBlockSize,
                    uint32_t capPtr);

static inline void PURE
cap_untyped_cap_ptr_new(cap_t *cap_ptr,
                        uint32_t capBlockSize,
                        uint32_t capPtr);

static inline uint32_t CONST
cap_untyped_cap_get_capBlockSize(cap_t cap);

static inline uint32_t PURE
cap_untyped_cap_ptr_get_capBlockSize(cap_t *cap_ptr);

static inline cap_t CONST
cap_untyped_cap_set_capBlockSize(cap_t cap,
                                 uint32_t v);

static inline void
cap_untyped_cap_ptr_set_capBlockSize(cap_t *cap_ptr,
                                     uint32_t v);
```

Fig. 6. Generated function prototypes for the example

proofs. The function of the generated proofs is not only to show implementation correctness, but also to provide sufficient reasoning power to allow any statement involving the generated functions to be rephrased in terms of simple operations on abstract, high-level types in the theorem prover. This means that we can avoid invoking bit manipulations and pointer dereferences when reasoning about the packed structures as part of a larger proof. We can instead reason about higher-level types, for which there is well established support.

This abstract representation is expressed in terms of Isabelle's record types, which behave much like struct or record constructs in typical programming languages, providing access and update of disjoint fields. The bitfields from the C level are represented as records of fields on the abstract level. For example, the `untyped_cap` block is represented as follows:

```
record cap_untyped_cap_CL =
    capBlockSize_CL :: "word32"
    capPtr_CL :: "word32"
```

Tagged unions are represented by an algebraic datatype wrapping the records corresponding to the component bitfields, with one constructor for each. Tag

fields are not included in the record representation, but are implied by the choice of constructor within the union type. Any bitfields which are empty after the removal of the tag field are represented simply by a naked constructor, with no associated record. The `cap` union translates thus:

```
datatype cap_CL =
    Cap_null_cap
  | Cap_untyped_cap cap_untyped_cap_CL
```

The name convention is that the C types and identifiers, when parsed into Isabelle, are tagged by appending `_C`, whereas the lifted types are tagged with `_CL`. The connection between the C level and the abstract level will be provided by two functions in the example below: *cap_lift* and *cap_untyped_cap_lift*. The former lifts any *cap_C* to a *cap_CL*, and the latter lifts a *cap_C* with the `untyped_cap` tag directly to a *cap_untyped_cap_CL*. It is under-specified in all other cases.

The properties of the generated functions are expressed as strongest-postcondition Hoare rules. Specifically, we use Schirmer's [23] verification environment for imperative programs in Isabelle/HOL. It contains a verification condition generator (VCG) which automates reasoning about Hoare-triples. Tuch's et al. [26] instantiation to C parses the generated code directly into Isabelle/HOL and into Schirmer's representation language SIMPL.

As an example, we will take the generated specification of the generated C function `cap_untyped_cap_set_capBlockSize`. It takes two arguments, an untyped capability *cap* and a new block size *v*. It returns the original capability updated with the new block size. The formal specification below translates this into a record update:

```
"Γ ⊢ ⦃s. cap_get_tag ´cap = cap_untyped_cap⦄
     ´ret__struct_cap_C :==
       PROC cap_untyped_cap_set_capBlockSize(´cap, ´v)
     ⦃cap_untyped_cap_lift ´ret__struct_cap_C =
      cap_untyped_cap_lift (s cap) (|capBlockSize_CL := (s v) AND (mask 5)|) ∧
      cap_get_tag ´ret__struct_cap_C = cap_untyped_cap⦄"
```

The specification above reads as follows. For all program contexts Γ, if the tag of the C-struct *cap* in the current state (indicated by ´) equals the value *cap_untyped_cap*, and we execute the function *cap_untyped_cap_set_capBlockSize* with parameters *cap* and *v*, storing its return value in *ret__struct_cap_C*, we will arrive at the following post condition: lifting the return value to the abstract record type is the same as lifting the value of *cap* in the initial state *s*, and then performing an update of the abstract record field *capBlockSize_CL* with the value *v* had in state *s*. Additionally, as a convenience for automated methods in the larger proof, we provide that the tag of the return value remains *cap_untyped_cap*. A separate specification states (and the tool proves) that the function is side-effect free, i.e. that no global variables, including the heap, are changed. The term *AND (mask 5)* carries the additional information that the field has a size of 5 bits. This form proved more convenient than the alternative of having an abstract field of word

length 5, because casting between word lengths often introduces additional proof obligations.

The meaning of the rule can also be expressed by means of the commuting diagram in Fig. 7.

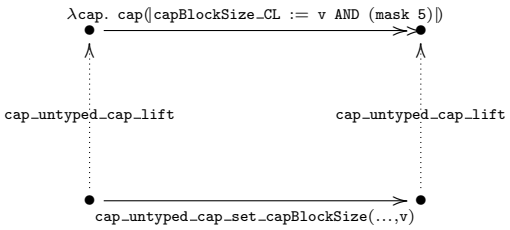


Fig. 7. Refinement picture for field update.

For Fig. 7, consider `cap_untyped_cap_set_capBlockSize` as a function from `cap_t` (its first argument) to `cap_t` (its return value). Control flows left to right, and *r* (| a := x |) is the Isabelle syntax for the record *r*, with field a updated with value *x*. This makes it clear that the function of the rule is to allow us to transform a function call into a record update, by commuting it with a lift. We can therefore take any precondition of the form *P (cap_untyped_cap_lift cap)*, and commute it past any number of field updates, to produce a postcondition of the form *P (f (cap_untyped_cap_lift cap))*, where *f* is the composition of a number of record updates.

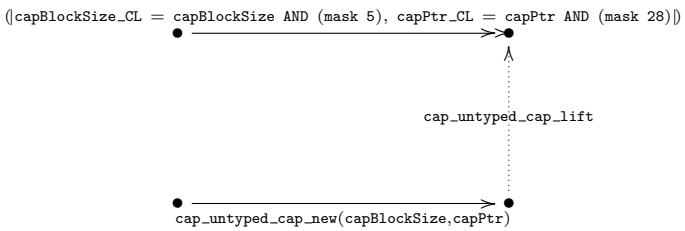


Fig. 8. Refinement picture for initialisation

That this is useful becomes clear when we consider the equivalent diagram for the *cap_untyped_cap_new* function (Fig 8), which returns a new untyped capability. This provides a starting point for our chain of reasoning, by providing the identity *cap_untyped_cap_lift cap = (|capBlockSize_CL=capBlockSize AND (mask 5),capPtr_CL=capPtr AND (mask 28)|)*. We can base our argument on such a case, as long as bitfield objects are only initialised via the appropriate `*_new` functions, and the type tags are never externally modified. This justifies the API restriction

introduced in Section 3, which is adhered to by the seL4 kernel implementation without loss of convenience or performance.

Equivalent rules are proved automatically for all the generated functions, and their pointer-lifted versions. The latter involve direct heap access to record fields and automate the interactive reasoning Tuch provides [25]. Additionally to what Tuch provides, we make use of the concept of packed records, which allow us to ignore padding in record implementations and derive more precise properties of the corresponding memory layout. Packed records are represented by a type class in Isabelle that simply states that all fields in the record are of a size that makes padding unnecessary.

The proofs of the specifications above are fully automated and generally consists of two to three automated method invocations in Isabelle. The first of these is a call to the C-level VCG mentioned above. The second and possibly third, first reduce the remaining proof obligation from variable, heap, and struct-updates to a goal on bit-vectors only. This is then solved automatically with a carefully designed set of generic, algebraic rewrite rules for the bit operations involved in the generated functions. The direct proof script for one specification of a typical C-function is typically about 10 lines long.

5 Related Work

Earlier work on OS verification includes PSOS [10] and UCLA Secure Unix [27]. Later, KIT [2] describes verification of process isolation properties down to object code level, but for an idealised kernel far simpler than modern microkernels. The Verisoft project [11, 12] is attempting to verify a whole system stack, including hardware, compiler, applications, and a simplified microkernel: VAMOS. The VFiasco [14] project is attempting to verify the Fiasco kernel, another variant of L4, directly at the C++ level. These verification projects do not use generated C code for automating parts of their proof obligations. In the case of Verisoft, there is no reason to distrust the compiler as it is also verified [20, 15]. Directly using C or dropping down to assembly code to implement the desired features does not have the benefit of the high-level reasoning support and API the tool presented here provides, however.

The verified proofs in this work build directly on Tuch's et al. memory model for C [26, 25, 24] which in turn builds on work by Schirmer [23, 22] that provides a generic framework, verification condition generator, and Hoare logic [13] for imperative programs. Both are intended for interactive verification. This paper uses the predictable structure of the generated code to completely automate the pointer level proofs on the C implementation.

This work also builds directly on Dawson's machine-word library [4] for Isabelle/HOL. Despite recent progress in tools like Yices [6], bit-vector proofs for machine words remain hard to automate. Traditional SAT solvers are usually too

slow to handle the resulting proof obligations on realistic word sizes. Again, due to the predictable nature of the generated code, the tool is able to fully automate the bit-level verification conditions with a set of algebraic rewrite rules. This means that switching to different, say 64-bit, architectures should not result in any noticeable slowdown of the generated proofs.

General translation validation [21] and compiler correctness including Leroy's et al. work [16] are related to the topic. As mentioned above, the tool presented here can exploit the known, predictable nature of the application domain to provide a convenient interface to, and integration into, the user's proofs.

Also related to generated correctness proofs is the idea of proof-carrying code [18], which usually focusses on the machine level and on specific properties such as memory safety or resource constraints. Functional correctness is not usually targeted, because it is impossible to automate completely. Barthe et al. [1] come close by automatically transforming certificates from source code to machine code and, similarly to the work presented here, generating proofs for generated code. In contrast to this, the bitfield generator here does not require any source-level proof as input. It generates a full functional correctness statement automatically.

Denney et al. [9] automatically prove properties about generated aerospace software. The generated code appears more complex than that presented here, but the semantics they are using is not foundational and the properties do not cover full functional correctness, only specific safety properties.

6 Conclusion

This paper has summarised a generator for tagged unions of packed bitfields in the C programming language, as they are used in low-level systems code and operating system kernel implementations. In theory, this data structure can be implemented with C primitives without resorting to a generator. However, compiler implementations of bitfields seem to be so unpredictable in memory layout and performance over different platforms and compilers that kernel programmers distrust this compiler feature more than others.

The tool generates efficient, predictable, and above all correct C code from a short, high-level description that is detailed enough to provide precise memory layout specification which is important to map data, for instance, to memory mapped hardware device registers. The generated code includes the data type itself as well as an API for convenient, high-level access on the stack and on the heap.

This work shows that an automatic correctness proof of generated code for controlled environments is not hard to achieve, even if this code contains bit-level reasoning and pointer access. The proof is foundational in the sense that it assumes no specific axioms on the application domain, but is built directly on

the semantics of the C programming language. The proof is machine-checked in the theorem prover Isabelle/HOL and provides an example of translation-validation: Instead of proving the correctness of the generator, the correctness of the generated code is proven instead.

The usefulness of the tool reaches further than a stand-alone correctness proof. The Hoare-triples proven integrate directly, within the same formal model, with larger implementation proofs of client code using the generated bitfields. The Hoare-triples are designed such that client proofs and code have no need to reason about the internal representation or bit-level operations that are carried out. They provide an abstract interface. Translation validation has likely made this easier to achieve than in a generator correctness proof. No meta-level reasoning or switching of formal models is required.

The tool is expected to automate 14% of the C implementation proofs in the L4.verified project, covering 1,800 lines of C code with 7,800 lines of generated proof. The seL4 kernel, including all generated inline functions, comprises 12,600 lines of code. The tool is generally applicable to code that needs to have direct, reliable control over the memory layout of data structures. The technique of generating proofs that integrate well into interactive environments should generalise easily to constrained application domains where the structure of the generated proof is predictable.

References

1. G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In *Proceedings of the 13th International Static Analysis Symposium (SAS)*, volume 4134 of *LNCS*, pages 301–317, Seoul, Korea, August 2006. Springer.
2. William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
3. David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Cesar Munoz and Otmane Ait, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, LNCS. Springer, 2008. To appear.
4. Jeremy E. Dawson. Isabelle theories for machine words. In *Seventh International Workshop on Automated Verification of Critical Systems (AVOCS'07)*, Electronic Notes in Computer Science, Oxford, UK, September 2007. Elsevier. To appear.
5. Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proc. ACM SIGPLAN Haskell Workshop*, Portland, OR, USA, September 2006.
6. Bruno Dutertre and Leonardo de Moura. The yices smt solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, 2006. Link visited June 2008.
7. Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. A memory allocation model for an embedded microkernel. In *Proc. 1st MIKES*, pages 28–34, Sydney, Australia, 2007.
8. Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.
9. Bernd Fischer Ewen Denney and Johann Schumann. Using automated theorem provers to certify auto-generated aerospace software. In *Proc. 2nd International Joint Conference on Automated Reasoning (IJCAR'04)*, volume 3097 of *LNCS*, pages 198–212. Springer, 2004.

10. Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conf. Proc., 1979National Comp. Conf.*, pages 329–334, New York, NY, USA, June 1979.
11. Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In Joe Hurd and Thomas F. Melham, editors, *Proc. TPHOls'05*, volume 3603 of *LNCS*, pages 1–16, Oxford, UK, 2005. Springer.
12. Mark A. Hillebrand and Wolfgang J. Paul. On the architecture of system verification environments. In *Hardware and Software: Verification and Testing*, volume 4899 of *LNCS*, pages 153–168, Berlin, Germany, 2008. Springer.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
14. Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *Proc. 2nd ECOOP-PLOS Workshop*, Glasgow, UK, October 2005.
15. Dirk Leinenbach and Elena Petrova. Pervasive compiler verification—from verified programs to verified systems. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings of the 3rd international Workshop on Systems Software Verification (SSV'08)*, Electronic Notes in Computer Science, Sydney, Australia, February 2008. Elsevier. To appear.
16. Xavier Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *33rd symposium Principles of Programming Languages (POPL'06)*, pages 42–54, New York, NY, USA, 2006. ACM.
17. J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.
18. George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.
19. Open Kernel Labs, Inc. OKL web site. http://www.ok-labs.com, 2007. Visited May 2008.
20. Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Computer Science Department, Saarbrücken, Germany, 2007.
21. Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *Proc. 4th Intl. Conf. Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.
22. Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, AI, and Reasoning*, volume 3452 of *LNAI*, pages 398–414. Springer, 2005.
23. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
24. Harvey Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School for Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2008.
25. Harvey Tuch. Structured types and separation logic. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings of the 3rd International Workshop on Systems Software Verification (SSV'08)*, Electronic Notes in Computer Science, Sydney, Australia, February 2008. Elsevier. To appear.
26. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, 2007. ACM.
27. Bruce Walker, Richard Kemmerer, and Gerald Popek. Specification and verification of the UCLA Unix security kernel. *Commun. ACM*, 23(2):118–131, 1980.

Model Stack for the Pervasive Verification of a Microkernel-based Operating System*

Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan

Saarland University, Computer Science Dept.
66123 Saarbrücken, Germany
{md11,jandb,sebastian}@wjpserver.cs.uni-sb.de

Abstract. Operating-system verification gains increasing research interest. The complexity of such systems is, however, challenging and many endeavors are limited in some respect: Some projects focus on a particular aspect like memory safety, not pursuing functional correctness. Others restrict their verification efforts to a single layer of software, assuming correctness of those below. Only few projects aim at pervasive formal verification of a computer system over several software layers.
In our paper, we present an approach to the formal specification of a microkernel-based operating system at several layers and glance on our verification experience with this model stack. From our experience, we conclude that pervasiveness entails more than just cumulative verification efforts on several layers. In fact, it is a challenging task to integrate models and proofs into a uniform, coherent theory.

1 Introduction

Software-verification tools have greatly improved in recent years. As a consequence, the verification of low-level software gains increasing research interest as a touch-stone for the industrial applicability of software verification in general. While many aspects of operating-system verification are currently studied, they are not integrated into a uniform, coherent theory. In our paper, we concentrate on the integration of several operating-system models in order to form a pervasive verification stack. We point out the integration problems and describe the specific design decisions that we made in order to support integration.

A characteristic problem of operating-system software is that hardware components become visible and the program functionality cannot be expressed in the pure semantics of a high-level language like C. Usually, this functionality is implemented in assembly and encapsulated in a small number of C functions, or *primitives*. Thus, only a small part of low-level code is indeed implemented in assembly while larger code portions are written in C and just use these primitives. From a model-theoretic point of view, such primitives extend the original C semantics. Another important problem related to operating systems regards the atomicity of operations in a concurrent setting. The processor might be interrupted in its current computation after each *assembly instruction* but a single C statement is usually compiled into several instructions.

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Consequently, we need an assembly semantics, a C semantics and a simulation theorem between both semantics. Leinenbach and Petrova [1] provide exactly that: a proven correct compiler that translates programs from the perfectly type-safe C variant *C0* into the assembly language of our RISC processor. An important feature of C0 is the inline-assembly statement, which permits the language extensions via primitives. Using this mechanism, we have implemented our operating system in C0. The correct C0 compiler is an elementary prerequisite for our model stack.

Outline. In Sect. 2, we explain the fundamentals of our operating system. This section comprises a sketch of the correct compiler, a survey of the features of our microkernel, and an outline of the system's implementation design. With these prerequisites in place, we give an overview of our model stack in Sect. 3. Certainly, we cannot descend into all details of the stack in this paper.[1] Hence, we confine ourselves to aspects of encapsulation and state partitioning. In Sect. 4, we describe our formal specification of the generic interface between the user processes and the kernel. Moreover, we sketch our formal model for assembly and C0 processes. This process abstraction is an elementary prerequisite for the model of *communicating user processes* as well as for the top-level specification of the operating system. We illustrate in Sect. 5, why we use the process abstraction already in the VAMOS specification and how we have constructed the latter model in order to support the abstraction towards COUP and the top-level specification. We conclude in Sect. 6.

2 Background

2.1 A Correct Compiler for an Extensible Language

In this section, we summarize work of Leinenbach and Petrova [1,2]. They provide a formally verified compiler that translates programs from the perfectly type-safe C variant C0 into the assembly language of a RISC processor. Restrictions of C0 in comparison to C are that expressions must be side-effect-free, all type conversions have to be made explicitly, and there is no pointer arithmetic. In spite of that, C0 still features dynamic memory allocation and inlined assembly.

The inline-assembly statement is an important feature for language extensions. Though the effect of this statement cannot be expressed in the formal C0 semantics, the compiler literally embeds the given assembly code into the compilation. Using this hook, we can implement C0 functions that use inline assembly in order to extend C0 by a functionality that cannot be expressed by other C0 statements.

[1] The complete Isabelle/HOL theories of the models as well as detailed documentation are made available at the Verisoft repository, see <http://www.verisoft.de/VerisoftRepository.html>

For C0 and for the assembly language, two small-step transition semantics have been formalized in Isabelle/HOL. Leinenbach has formally proven compiler correctness by a stepwise simulation between both semantics. Below, we describe the semantics in detail. We often deal with structured values, which we define by enumerating the components in prose, e. g., "a value x consisting of two components *this* and *that*". We refer to a single component with a dot, e. g., $x.\mathit{this}$ refers to component *this* of value x. An update of this component is denoted by $x\,[\mathit{this} := q]$.

C0 Semantics. For lack of space, we can only glance at the C0 semantics. C0 programs are statically represented by the program environment Γ, which comprises a symbol table of global variables, a type-name environment, and a function table. The dynamically changing state s_{C0} of a C0 program in execution is composed of

- the remaining program $s_{\mathrm{C0}}.\mathit{prog}$, and
- the current state of the program variables $s_{\mathrm{C0}}.\mathit{mem}$.

In the following sections, we assume an evaluation function *get-val* for the look-up, and an update function *set-val* for the manipulation of a certain variable in the memory. We refer to the value of expression e in state s_{C0} by $\mathit{get\text{-}val}(s_{\mathrm{C0}}, e)$. If we update the left-value l in state s_{C0} with some expression u, we denote the resulting configuration by $\mathit{set\text{-}val}(s_{\mathrm{C0}}, l, u)$.

The transition relation $\delta^{\mathrm{seq}}_{\mathrm{C0}}$ of this semantics is a partial function.

The Target Assembly Language. The assembly semantics was developed for the RISC processor VAMP [3]. The assembly semantics abstracts from the paging mechanism of the processor and employs a linear memory model. An assembly state s_{asm} consists of the following components:

- the normal and the delayed program counters, $s_{\mathrm{asm}}.\mathit{pc}$ and $s_{\mathrm{asm}}.\mathit{dpc}$, respectively, implementing the delayed branch mechanism.
- the general-purpose register file $s_{\mathrm{asm}}.\mathit{gpr} \in \{0,\ldots,31\} \to \{0,1\}^{32}$.
- the memory size $s_{\mathrm{asm}}.V$ measured in pages of 4096 bytes. It defines the set of available memory addresses: $\mathit{VA}(s_{\mathrm{asm}}) = \{a \mid a < s_{\mathrm{asm}}.V \cdot 4096\}$
- the byte-addressable linear memory $s_{\mathrm{asm}}.\mathit{vm} \in \mathit{VA}(s_{\mathrm{asm}}) \to \{0,1\}^8$

We denote the state space of the assembly semantics by $\mathcal{S}_{\mathrm{asm}}$. Assembly computations are modeled by the partial function $\delta^{\mathrm{seq}}_{\mathrm{asm}} \in \mathcal{S}_{\mathrm{asm}} \rightharpoonup \mathcal{S}_{\mathrm{asm}}$. Note that the effects of exceptions like illegal page faults cannot be fully determined from the assembly-machine state. In that case, $\delta^{\mathrm{seq}}_{\mathrm{asm}}$ gets stuck. But with sufficient resources, a compiled C0 program does not generate exceptions during normal execution. Moreover, the memory size $s_{\mathrm{asm}}.V$ can neither be read nor changed by the assembly machine itself but depends on the operating-system kernel. We extend the semantics accordingly in Sect. 4.

2.2 Fundamentals of our Microkernel

In this section, we sketch the features of VAMOS and explain the fundamental access-control mechanism that establishes the process roles "privileged process" and "device driver".

Our microkernel VAMOS performs the following tasks: (a) enforcement of a minimal access control, (b) process management, (c) memory management, (d) priority-based round-robin scheduling, (e) support for user-mode device drivers, and (f) inter-process communication (IPC). Processes can control these tasks via the kernel's application binary interface (ABI). Table 1 lists the kernel calls that constitute the ABI.

Most kernel calls are reserved for so-called *privileged processes*. Thus, only a privileged process can bring up new processes or kill existing ones, alter the memory consumption of processes, change their scheduling parameters, or control the registration of device drivers. Any process, however, might use the IPC mechanism. Thus, the privileging serves as a minimal access control. We presume that the privileged processes constitute the user-mode parts of the operating system and implement a more sophisticated access-control mechanism. Non-privileged processes may then communicate with the privileged processes in order to request kernel services on their behalf.

Table 1. Application binary interface of the VAMOS kernel

Kernel Call	**Description**
Access Control	
`set_privileged` [p]	add a process to the set of privileged processes
Process Management	
`process_create` [p]	create a new process from a memory image
`process_clone` [p]	copy an already existing process
`process_kill` [p]	kill a process
Memory Management	
`memory_add` [p]	increase the amount of virtual memory for a process
`memory_free` [p]	decrease the amount of virtual memory for a process
Scheduling Mechanism	
`chg_sched_params` [p]	change scheduling parameters
Device Driver Support	
`change_driver` [p]	(un)register a process as a driver for a set of devices
`enable_interrupts` [d]	re-enable a set of interrupts after their successful handling
`dev_read` [d] / `dev_write` [d]	communicate with a certain device
Inter-Process Communication	
`ipc_send`	send a message to another process
`ipc_receive`	receive a message from another process
`ipc_request`	send a message and immediately wait for a reply
`change_rights`	manipulate IPC rights
`read_kernel_info`	receive information from the kernel

[p] call is reserved for privileged processes [d] call is reserved for device drivers

When VAMOS boots, it launches one single process, the *init process*. This process is privileged and has to set up the required servers of the operating system, start and register the device drivers, and possibly run initial applications.

A *device driver* is a user process, which is designated for the communication with certain devices. Only if a process is registered as a driver for a particular device, it may place read or write requests from or to that device, respectively. Moreover, the device driver is notified of interrupts from that device.

2.3 Implementation Design

Fig. 1 depicts the implementation structure of VAMOS. The lowest software layer is called *communicating virtual machines* (CVM). This layer encapsulates all hardware-specific low-level functionality, which is possibly using inline assembly. CVM has two major tasks: memory virtualization and switching between different threads of execution. Hence, CVM includes a page-fault handler with a simple memory swapping facility [4]. Moreover, it exports an interface of so-called *primitives* for the access and manipulation of user machines to the hardware-independent part. We have thereby established a solid framework for microkernel construction.

Using this framework, we have implemented our microkernel VAMOS in C0 without extra portions of inline assembly. On every kernel entry, CVM preserves the old context, establishes a suitable C0 environment and calls the function `kdispatch` of VAMOS. For the manipulation of the user memory or registers, VAMOS may call the primitives of CVM. The return value of `kdispatch` determines, which user machine resumes when the kernel execution finishes.

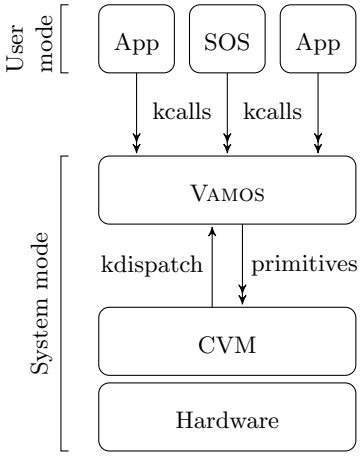


Fig. 1. Implementation scheme

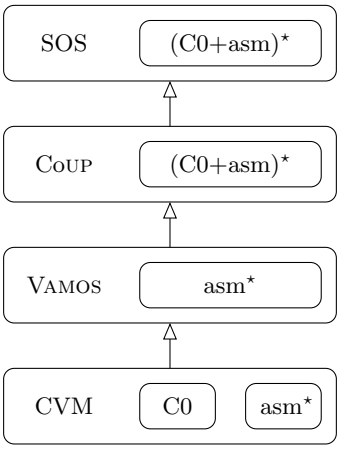


Fig. 2. Schematics of our model stack

While CVM and Vamos run in the privileged system mode of the processor, user machines run in the unprivileged user mode. In the figure, we labelled one user machine "SOS" for *simple operating system* and the others "App" as abbreviation for application. The SOS constitutes the highest layer of our operating system. It features an advanced rights management with different users, implements a sophisticated access control to kernel services like process creation and provides further services like file-system and network access. All user machines interact with the kernel via *kernel calls*. The special instruction `trap` causes an exception, which is handled in Vamos. Vamos can examine and alter the state of the user machine using CVM primitives, thus identifying the user machine's specific request and storing the kernel's corresponding response.

3 Model-Stack Overview

In analogy to the software stack, we have developed a model stack. For each software layer, this stack contains a specification that fully describes the observable behavior of this layer. The specification describes the full functionality of the software layer, on the one hand, and forms the foundation for the simulation proofs for the more abstract layers, on the other hand. We build the model stack as a means of verifying statements about the real system on a convenient, abstract layer. Such statements include safety and liveness properties.

Fig. 2 shows our model stack. The lowest model describes CVM. This model comprises a kernel machine, given in the C0 semantics, and a number of user machines with virtual memory, given in the assembly semantics. The kernel runs in system mode with disabled interrupts. The user machines, in contrast, may be interrupted after each assembly instruction, thus we model them on assembly level. We will not descend into the details of this model in the present publication because it has been described earlier [5].

We can instantiate the kernel of CVM with the Vamos implementation, i. e., the C0 program that implements (the hardware-independent part of) our microkernel is executed as CVM's kernel machine. The Vamos model specifies the behaviour of the resulting system. In short, this model establishes assembly processes that communicate with the kernel via a well-defined interface, the kernel ABI. A code-correctness proof for the Vamos implementation establishes the simulation theorem between the instantiated CVM model and the Vamos specification.

While the Vamos processes use the assembly semantics, the SOS and the applications are written in C0. The kernel ABI is encapsulated in a few library functions. Except for the library functions, the verification of these programs should certainly employ the more abstract C0 semantics. For this purpose, we designed the model *communicating user processes* (Coup). This model simulates Vamos but it abstracts from the scheduler and establishes C0 processes.

We need to abstract from the particular VAMOS scheduler because of the different granularity of the program semantics: The execution of a process might be suspended and resumed on every assembly instruction but a C0 statement usually contains several assembly instructions. The simulation proof has three parts: (1) The execution traces of COUP should contain all traces of VAMOS. (2) There is a confluent reordering of any execution trace such that there is no scheduling event during a single C0 statement. (3) The kernel library is correctly implemented.

We can instantiate one of the C0 processes with the implementation of SOS. The SOS model specifies the behaviour of the resulting system. The simulation between both models can be shown by the functional correctness of the SOS implementation.

4 Process Abstraction

In this section, we formalize the interface between the processes and the kernel. Both kernel models, the VAMOS specification and the COUP model, use this process abstraction in order to access and manipulate processes. Our abstraction is based on the observation that VAMOS interacts with processes only via a well-defined interface, the kernel ABI. Hence, we can encapsulate processes in a self-contained input-output automaton, thereby hiding the internal state and just exposing the generic interface.

We define a process as input-output automaton described by a tuple

$$(\mathcal{S}_{\text{proc}}, \Sigma_{\text{proc}}, \Omega_{\text{proc}}, \omega_{\text{proc}}, \mathit{vm\text{-}size}_{\text{proc}}, \mathit{init}_{\text{proc}}, \delta_{\text{proc}})$$

with state space $\mathcal{S}_{\text{proc}}$, input alphabet Σ_{proc}, output alphabet Ω_{proc}, output functions ω_{proc} and $\mathit{vm\text{-}size}_{\text{proc}}$, initialization function $\mathit{init}_{\text{proc}}$, and transition function δ_{proc}.

While the state space $\mathcal{S}_{\text{proc}}$ depends on the individual process abstraction, the interface between the kernel and the processes is naturally shared by all process abstractions. This interface is entirely defined by Σ_{proc} and Ω_{proc}.

The output alphabet Ω_{proc} enumerates all possible kernel calls. Additionally, we have to treat a few error cases. As the kernel calls are internally identified by a number, a process might specify an invalid number. This condition is represented by the special output value undefined_trap. Moreover, a process might generate exceptions like an arithmetic overflow or an illegal page fault. These exceptions are collectively represented as value runtime_error. Finally, the output ε denotes the intention to perform a local computation.

The input alphabet Σ_{proc} reflects all kernel-initiated changes of a process. These comprise all possible responses to kernel calls, on the one hand, and the demand to change the amount of virtual memory, on the other hand. While the

former are the synchronous reaction to a kernel call, the latter may be issued asynchronously at any stage of a process. In order to perform a local transition, we pass the input ε to the transition function δ_{proc}.

In addition to the usual functions $\delta_{\text{proc}} \in \Sigma_{\text{proc}} \times \mathcal{S}_{\text{proc}} \to \mathcal{S}_{\text{proc}}$ for transitions and $\omega_{\text{proc}} \in \mathcal{S}_{\text{proc}} \to \Omega_{\text{proc}}$ for the output, we use two more functions in our process abstraction. In order to compute the overall memory consumption of the process system, VAMOS needs to know the amount of virtual memory that is currently occupied by every process. The function $\mathit{vm\text{-}size}_{\text{proc}} \in \mathcal{S}_{\text{proc}} \to \mathbb{N}$ provides the necessary information. When a new process is created, VAMOS has to translate a representation of a given binary executable file into the corresponding, initial process state. We specify this translation in the function $\mathit{init}_{\text{proc}} \in \mathbb{N}^\star \to \mathcal{S}_{\text{proc}}$.

For the kernel specification, the generic abstraction of processes as self-contained input-output automata is rather a matter of taste than necessity. However, this generic process abstraction is a cornerstone of the CoUP model. Moreover, it is crucial for the simulation theorem between VAMOS and CoUP because both models mainly differ in the process abstraction. Hence, it proved to be beneficial that many definitions of the specification are parametrized over the process abstraction and can thus be reused in CoUP as well. Finally, the introduction of this parameter on the whole CoUP model is just a logical consequence because it paves the way for the integration of other language semantics.

In the following, we refine the generic abstraction with specific process models for assembly and C0.

Assembly Processes. We reuse the state space $\mathcal{S}_{\text{asm}}$ of the assembly semantics for our assembly processes. Based on this state space, we now define the output functions ω_{asm} and $\mathit{vm\text{-}size}_{\text{asm}}$, the transition function δ_{asm}, and the initialization function $\mathit{init}_{\text{asm}}$. The function $\mathit{vm\text{-}size}_{\text{asm}}$ simply returns the value of the component V of the current state: $\mathit{vm\text{-}size}_{\text{asm}}(s_{\text{asm}}) = s_{\text{asm}}.V$. However, the definitions for other functions are too complex to fully present them here. We constrain ourselves to an exemplary excerpt.

Fig. 3 depicts the formal definition of the output function ω_{asm} and the transition function δ_{asm} for the call `process_clone`. We assume that s_{asm} is the state of an assembly process. The predicate *trap* holds iff the current instruction is a trap, and the function *simm* extracts the sign-extended immediate constant from the current instruction. If there is a trap with immediate constant 2, the output function will return the pair of value process_clone and the content of register 11. Let us now assume that the kernel recognizes this output from the current process but the process is not privileged. In this case, the kernel signals this error condition by passing value err_unprivileged via the transition function to the current process. Then, the transition function updates the register 22 with the corresponding error code and increases the program counters.

$$trap(s_{\text{asm}}) \wedge simm(s_{\text{asm}}) = 2 \implies \omega_{\text{asm}}(s_{\text{asm}}) = (\mathsf{process\_clone}, s_{\text{asm}}.gpr(11))$$

$$\delta_{\text{asm}}(\mathsf{err\_unprivileged}, s_{\text{asm}}) = s_{\text{asm}} \left[\begin{array}{rcl} gpr(22) & := & -4 \\ pc & := & s_{\text{asm}}.pc + 4 \\ dpc & := & s_{\text{asm}}.dpc + 4 \end{array} \right]$$

Fig. 3. Formal definition of the output function and the transition function of assembly processes for the call `process_clone`

C0 Processes. In order to represent C0 processes, we define an automaton with the following signature:

$$(\mathcal{S}_{\text{C0}}, \Sigma_{\text{proc}}, \Omega_{\text{proc}}, \omega_{\text{C0}}, \mathit{vm\text{-}size}_{\text{C0}}, \mathit{init}_{\text{C0}}, \delta_{\text{C0}})$$

Any state $s_{\text{C0}} \in \mathcal{S}_{\text{C0}}$ comprises the static program environment as well as the dynamic program state of the C0 machine as described in Sect. 2.1. We store the program environment because processes might dynamically be created and killed, hence the program might change. The output functions ω_{C0} and $\mathit{vm\text{-}size}_{\text{C0}}$, the transition function δ_{C0}, and the initialization function $\mathit{init}_{\text{C0}}$ are defined in analogy to their assembly-process counterparts.

For example, the formal definition of the output function and the transition function for the call `process_clone` is given in Fig. 4. We assume that s_{C0} is the current state of a C0 process. The component $s_{\text{C0}}.\mathit{prog}$ denotes the remaining program of the process. We consider the first statement of the program. If this statement is a call to the function `vc_process_clone`, we define the output of ω_{C0} as the pair of process_clone and the value of the first argument to the function call. Let us now assume that the kernel recognizes this output from the current process, clones the given process, and computes the value hn_{new} as new identifier for the clone. It would then pass this value via the transition function to the current process, thus signalling the success of the clone operation. In this case, the transition function updates the memory $s_{\text{C0}}.\mathit{mem}$ at the address designated by the left-value e with value hn_{new} and removes the function call from the remaining program.

5 Formal Kernel Models

In the previous section, we described our process model. Now, we embed the processes into the two concurrent kernel models, the VAMOS specification, and the COUP model. The former specifies the exact behaviour of our microkernel with a particular scheduler. This model is used for code verification. The latter abstracts the scheduler and focusses on the interaction of the processes with the

$$s_{\mathrm{C0}}.prog = \text{"}e = \texttt{vc\_process\_clone}(e_0); r\text{"} \implies \omega_{\mathrm{C0}}(s_{\mathrm{C0}}) = (\mathsf{process\_clone}, \mathit{get\text{-}val}(s_{\mathrm{C0}}.mem, e_0))$$

$$s_{\mathrm{C0}}.prog = \text{"}e = \texttt{vc\_process\_clone}(e_0); r\text{"} \implies \delta_{\mathrm{C0}}((\mathsf{succ\_new\_process}, hn_{\mathrm{new}}), s_{\mathrm{C0}}) = s_{\mathrm{C0}}\begin{bmatrix} mem & := & \mathit{set\text{-}val}(s_{\mathrm{C0}}.mem, e, hn_{\mathrm{new}}) \\ prog & := & r \end{bmatrix}$$

Fig. 4. Formal definition of the output function and the transition function of C0 processes for the call `process_clone`

microkernel.We need this more abstract model in order to describe the reordering of interleaved sequences and to introduce C0 processes.

Both models are Moore machines. We describe the kernel specification with the tuple $(\mathcal{S}_{\mathrm{v}}, s^0_{\mathrm{v}}, \hat{\Sigma}, \hat{\Omega}, \omega_{\mathrm{v}}, \delta_{\mathrm{v}})$ and the CoUP model with $(\mathcal{S}_{\mathrm{vc}}, s^0_{\mathrm{vc}}, \hat{\Sigma}, \hat{\Omega}, \omega_{\mathrm{vc}}, \delta_{\mathrm{vc}})$. The state spaces $\mathcal{S}_{\mathrm{v}}$, $\mathcal{S}_{\mathrm{vc}}$ contain the initial states s^0_{v} and s^0_{vc}, respectively. For device communication, we use the input alphabet $\hat{\Sigma}$ and the output alphabet $\hat{\Omega}$. The functions ω_{v} and ω_{vc} determine the output in a current state. Finally, δ_{v} and δ_{vc} describe the transitions of the models. The notable difference between these machines regards the determinism: While the VAMOS specification is fully deterministic, CoUP features a non-determinism in its scheduling decisions. Consequently, the transition relation δ_{v} is functional while δ_{vc} is not.

Below, we introduce the different components of the models side by side.

Device Communication. Our kernel uses memory-mapped I/O for device communication. Hence, the output alphabet $\hat{\Omega}$ comprises read and write accesses to device addresses. The input alphabet $\hat{\Sigma}$ consists of interrupt lines and optionally incoming data. Hillebrand *et al.* [6] have described our device interface in detail.

State Spaces. A state $s_{\mathrm{v}} \in \mathcal{S}_{\mathrm{v}}$ comprises the following components:

- The partial function $s_{\mathrm{v}}.procs$ maps the process identifiers (PIDs) of the currently active processes to their assembly states $s_{\mathrm{asm}} \in \mathcal{S}_{\mathrm{asm}}$. For inactive processes, this function is undefined.
- Priorities are assigned to each PID of the active processes with the partial function $s_{\mathrm{v}}.priodb$.
- All other scheduling information is kept in the component $s_{\mathrm{v}}.schedds$.
- The partial function $s_{\mathrm{v}}.rightsdb$ maps PIDs to a data structure for the management of IPC rights and the set of privileged processes.
- Finally, the component $s_{\mathrm{v}}.devds$ contains data for device communication.

At first sight, the separation of priorities from the remaining scheduling data might surprise, here. The reason for this unintuitive partitioning is that CoUP abstracts from the particular scheduling algorithm but the priorities remain visible. We modularized the states and the corresponding transition functions in

order to share the definition of similar parts between the VAMOS specification, CoUP, and the SOS model.

A central part of the VAMOS specification are the scheduling data structures. The component $s_{\mathrm{v}}.\mathit{schedds}$ is divided into sub components. The current time $\mathit{time} \in \mathbb{N}$ is a counter for clock ticks. Process-specific scheduling information for active processes is collected in the partial function *procdb* that maps PIDs to a record of (a) the time slice *tsl*, (b) the amount of consumed time *ctsl*, and (c) the absolute timeout *to*. If a process is found to be computing when a timer interrupt raises, the component *ctsl* is increased until the process has finally run for *tsl* ticks. In this case, another process is scheduled. If a process calls the kernel for IPC and no partner is ready for communication, the absolute timeout *to* is computed from the current time and the relative timeout that has been specified with the call.

Moreover, the scheduler maintains different queues for scheduling. They are represented as finite sequences in the VAMOS specification. Namely, there is a ready queue $\mathit{ready}(\mathit{prio})$ of schedulable processes for each priority $\mathit{prio} \in \{0, 1, 2\}$. The processes that cannot currently be scheduled (because they are waiting for an IPC partner) are held in a queue named *wait*.

In VAMOS, the current process is the first process in the highest, non-empty ready queue. If all ready queues are empty, the current process is undefined. Formally, we define the function *cup* as:

$$\mathit{cup}(s_{\mathrm{v}}.\mathit{schedds}) = p \iff \exists i : s_{\mathrm{v}}.\mathit{schedds}.\mathit{ready}(i) = (p, \dots) \wedge \forall\, j > i : s_{\mathrm{v}}.\mathit{schedds}.\mathit{ready}(j) = ()$$

The corresponding CoUP state s_{vc} inherits most components of s_{v}. Only two components change: The process abstraction becomes a model parameter, i. e., component $s_{\mathrm{vc}}.\mathit{procs}$ of a CoUP state s_{vc} is a partial function from PIDs to a generic state space $\mathcal{S}_{\mathrm{proc}}$ for processes. Moreover, the scheduling data structures are replaced by a current-process indicator. We retain the current process in the state in order to compute the output from the current state. The output function ω_{vc} signals the demand for device communication. In order to determine this demand, we need to employ the output function ω_{proc} of the current process. Consequently, we fix this process beforehand instead of including transitions for all ready processes in the transition relation.

Transitions. A transition $\delta_{\mathrm{v}}(\hat{\sigma}, s_{\mathrm{v}})$ of the VAMOS specification under the device input $\hat{\sigma} \in \hat{\Sigma}$ has up to three phases:

1. If the current process $\mathit{cp} = \mathit{cup}(s_{\mathrm{v}}.\mathit{schedds})$ is defined, we consult its output $\omega_{\mathrm{proc}}(s_{\mathrm{v}}.\mathit{procs}(\mathit{cp}))$ and compute the response according to the current VAMOS state. For instance, if a process calls `process_clone`, we check for sufficient privileges and resources and choose the corresponding response

$\sigma \in \Sigma_{\text{proc}}$ for success or failure. With this response, we advance the current process: $s_{\text{v}}\,[procs(cp) := \delta_{\text{proc}}(\sigma, s_{\text{v}}.procs(cp))]$.
2. If the timer-interrupt line is raised, the scheduler increases the clock-tick counter $s_{\text{v}}.schedds.time$ and the consumed time $s_{\text{v}}.schedds.procdb(cp).ctsl$ of the current process. Moreover, the scheduler wakes up all processes p with elapsed timeouts.
3. Finally, VAMOS delivers interrupts to waiting drivers and saves the remaining interrupts for later delivery in $s_{\text{v}}.devds$.

The COUP transitions behave very similarly. However, a transition $s_{\text{vc}} \xleftrightarrow{\sigma} s'_{\text{vc}} \in \delta_{\text{vc}}$, obtains cp directly from $s_{\text{vc}}.cup$. Moreover, the only visible effect of the second phase is the wake-up of certain waiting processes. This effect is simulated non-deterministically and independently from the timer interrupt.

Reusability. As mentioned earlier, we worked hard to reuse as much definitions as possible in the different layers of abstraction. For example, in contrast to the COUP model, the kernel specification does not actually rely on the encapsulation of processes as self-contained input-output automata. Since VAMOS only considers assembly processes, it would be easier to directly manipulate the state of these processes. However, in order to literally reuse update functions and definitions, we invested the extra work and already introduced the input-output automata at this level of our model stack.

Consider the formal definition of the function *v_clone_updt_procs* in Fig. 5. This function describes the necessary updates in case of a successful call to `process_clone`. Without going into detail, depending on the relation between the calling process (p_{subj}) and the process that was cloned (p_{obj}), this function updates the state of the calling process, the state of the cloned process, and the state of the new process. The important thing about this definition is that the updates are presented in a process-abstraction independent manner. Rather than updating particular registers, as it would suffice for assembly processes, we feed the generic transition function δ_{proc} with generic input, e.g. $(\mathsf{succ\_new\_process}, hn_{\text{new}})$. Now, this generic interface between kernel and processes allows us to use *v_clone_updt_procs* in both models, i.e. VAMOS *and* COUP. Proceeding in a similar way with all kernel calls keeps the models clear and maintainable. Furthermore, tedious equivalence proofs between VAMOS and COUP are avoided.

6 Conclusion

Related Work. With the CLI stack [7], a pioneering approach started out for pervasive verification of a complete system with rigorous formalizations of specifications and proofs. Most notably, the simple kernel KIT [8] was developed

$$
\begin{array}{l}
v\_clone\_updt\_procs(procs, p_{\text{subj}}, p_{\text{obj}}, p_{\text{new}}, hn_{\text{new}}) = \\
\quad \lambda x. \\
\qquad \textbf{if } x = p_{\text{subj}} \textbf{ then } \delta_{\text{proc}}((\mathsf{succ\_new\_process}, hn_{\text{new}}), procs(p_{\text{subj}})) \\
\qquad \textbf{else if } x = p_{\text{new}} \wedge p_{\text{subj}} = p_{\text{obj}} \textbf{ then } \delta_{\text{proc}}((\mathsf{succ\_new\_process}, 0), procs(p_{\text{subj}})) \\
\qquad \textbf{else if } x = p_{\text{new}} \textbf{ then } procs(p_{\text{obj}}) \\
\qquad \textbf{else } procs(x)
\end{array}
$$

Fig. 5. Formal definition of the function *v_clone_updt_procs* that computes the component *procs* after a call to `process_clone`

and its machine code implementation was proven to be correct. However, this kernel is fairly simple compared to modern microkernels.

Many recent projects undertake verification efforts on modern microkernels. Among them are L4.verified [9], VFiasco [10], Eros [11], and the Flint project [12]. Though these projects may have achieved some advances, they all discard pervasiveness but focus on the verification of a single software layer. The Flint project has developed a verification framework for an assembly language and has formally proven the correctness for context-switching code. The other three projects have established semantics for C variants and have verified different properties on source code level. As far as we can see, inlined assembly portions are just postulated to be correct and solely described by their semantical effects. Moreover, these projects rely on compiler correctness.

Traditionally, concurrent systems are described by the *Calculus of Communicating Systems* [13] or as *Communicating Sequential Processes* (CSP) [14]. Both approaches, however, argue on a very abstract level and are mainly used to specify system requirements at an early stage in the system design. Basin, Olderog and Sevinç [15], for instance, describe the combination of CSP and Object-Z in order to specify and analyze security automata. Unfortunately, a link to the actual code is missing.

Discussion. We presented an approach to a pervasive, formal specification of a microkernel-based operating system. The formal models of our microkernel Vamos and the SOS occupy almost 400 kB and comprise about 8,000 lines of specification. The formalization took us about 48 person months. The different specification layers resemble essentially the implementation layers.

A recurring problem was the semantic gap between the high-level implementation language C0 and certain hardware details that were manipulated by the considered programs. Such details concerned hidden hardware components like devices, on the one hand, and the granularity of atomic operations on the hardware level, on the other hand. Both problems arose simultaneously when we argued about C0 processes: In pure C0, we cannot express kernel calls. Hence, we extended the original C0 semantics by a library of functions that implement the

kernel calls using inlined assembly. Moreover, user machines may be interrupted after the execution of any assembly instruction. Consequently, we had to justify that we may confluently shift all rescheduling events to statement boundaries before we could argue about C0 processes. In non-pervasive approaches, these challenges are likely to be skipped.

When we began to specify the different layers of our operating system, we started with independent models. As expected, many parts of the formal models overlapped because we specified the same operating system at several layers of abstraction. After a short period of evaluation for each abstraction layer, we coupled the models as tight as possible in order to minimize the proof efforts for the simulation theorems between adjacent models.

This ambivalent approach was very successful. In the very beginning, the specifications change very often, such that maintaining dependencies between them would be very tedious and distracting from the individual problems. As soon as the models stabilized, however, there was a common ground for a tight meshing. The synergies between the layers could easily be identified and utilized.

The key to shared definitions between the VAMOS specification and the COUP model was twofold: On the one hand, we unitized the state space in a way that permits easy abstraction between the different abstraction layers. On the other hand, we established the process abstraction already at the VAMOS specification. Both arrangements together enabled a very tight mesh, which could even be reused in the SOS specification.

Of course, this tight mesh of the models has drawbacks: The specification of an abstract layer depends massively on the underlying layers, which impedes the understanding of this model. Moreover, even the lower layers became more complicated, so for instance, when the process abstraction was already introduced in the VAMOS specification. Finally, several layers were affected whenever the abstraction at one level turned out to be wrong.

From our experience, however, we conclude that the advantages outweigh the drawbacks. We saved valuable time by reusing the definitions, which we otherwise had spent in distracting equivalence proofs. For instance, we succeeded with the step-wise simulation proof between the VAMOS specification and the COUP model within two months. We have proven this simulation for most kernel calls within days. Most of the time, however, we spend with the verification of IPC. This effort was caused by a considerably simpler modelling of IPC in COUP, which became possible after the scheduler was abstracted. In this particular case, the simpler modelling was desirable and hence, we invested the additional time for the proof. Independent models, however, would possibly have caused such an overhead for every kernel call.

In the code-correctness proof for VAMOS, there is only a minimal overhead (hours) induced by the use of the process abstraction in the specification. All in all, we invested about 18 person months to show the correctness of the IPC send

call. Including all auxiliary functions, the implementation of this call accounts for nearly half the code size and comprises the most complex parts. Again 18 person month had been spent on the functional verification of the file system and the hard disk driver code in the SOS. This code covers 20% of the SOS implementation.

Summary. The fundamental difference of our "pervasive" approach compared to "incremental" approaches that focus on a single layer at a time regards the design of specifications: At first, models are usually adapted to the actual verification goal. If it is the only goal, to show that the formal specification precisely describes the implementation, the specification tends to move closely towards the implementation. If a model is used as fundament of an underlying component without a proof, there is a likelihood that the model over-abstracts the actual implementation. At second, regarding one single layer at a time necessarily leads to self-contained, independent models. When combining those different layers later on, there is a tremendous proof effort necessary to establish simulation theorems between the adjacent layers. Our approach prevents this effort by the specification design.

References

1. Leinenbach, D., Petrova, E.: Pervasive compiler verification: From verified programs to verified systems. In: Workshop on Systems Software Verification, to appear, Elsevier (2008)
2. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: SEFM. (September 2005) 2–11
3. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together: Formal verification of the VAMP. STTT **8**(4-5) (2006) 411–430
4. Alkassar, E., Starostin, A., Schirmer, N.: Formal pervasive verification of a paging mechanism. In: TACAS. (2007)
5. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In: TPHOLs. Volume 3603 of LNCS., Springer (2005) 1–16
6. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD, IEEE Computer Society (2005) 309–316
7. Bevier, W.R., Hunt, Jr., W.A., Moore, J.S., Young, W.D.: An approach to systems verification. J. Autom. Reasoning **5**(4) (December 1989) 411–428
8. Bevier, W.R.: Kit and the short stack. J. Autom. Reasoning **5**(4) (December 1989) 519–530
9. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: taking microkernels to the next level. Operating Systems Review (July 2007) 41(3)
10. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel: the vfiasco project. In: Operating Systems Review, European workshop: beyond the PC, New York, NY, USA, ACM Press (2002) 165–169
11. Shapiro, J.S., Weber, S.: Verifying the EROS confinement mechanism. In: Symposium on Security and Privacy, IEEE Computer Society (2000) 166–176
12. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: TPHOLs, Lecture Notes in Computer Science (September 2007) 189–206
13. Milner, R.: A Calculus of Communicating Systems. Springer (1982)
14. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
15. Basin, D.A., Olderog, E.R., Sevinç, P.E.: Specifying and analyzing security automata using CSP-OZ. In: ASIACCS, ACM (2007) 70–81

Exploring Model-Based Development for the Verification of Real-Time Java Code

Niusha Hakimipour[1], Paul Strooper[1], and Roger Duke[1]

University of Queensland, St. Lucia, Queensland, Australia
niusha@itee.uq.edu.au, pstroop@itee.uq.edu.au, rduke@itee.uq.edu.au

Abstract. Many safety- and security-critical systems are real-time systems and, as a result, tools and techniques for verifying real-time systems are extremely important. Simulation and testing such systems can be exceedingly time-consuming and these techniques provide only probabilistic measures of correctness. There are a number of model-checking tools for real-time systems. However, they provide formal verification for models, not programs. To increase the confidence in real-time programs written in real-time Java, this paper takes a modelling approach to the design of such programs. First, models can be mechanically verified, to check whether they satisfy particular properties, by using current real-time model-checking tools. Then, programs are derived from the model by following a systematic approach. To illustrate the approach we use a nontrivial example: a gear controller.

1 Introduction

Real-time [1] is a broad term used to describe applications that have timing requirements. Many safety- and security-critical systems are real-time systems and, as a result, tools and techniques for verifying real-time systems are extremely important. The traditional ways of ensuring that real-time systems operate correctly have been simulation and testing. However, in many cases these techniques are exceedingly time-consuming and provide only probabilistic measures of correctness. Formal methods advocate the use of mathematical reasoning as an alternative; one of the most promising of these methods has been model-checking [2].

There are a number of model-checking tools for real-time systems [3–6]. However, they provide formal verification for models, and no systematic approach for deriving programs from those models. This means it is still necessary to show that the programs that implement those models satisfy the properties as well.

Real-time systems have to generate their output within a finite and predicable time. Therefore, the specification of the language in which real-time systems are implemented is as important as verifying such systems. Real-Time Specification for Java (RTSJ) [1] was proposed in January 2002. Sun has developed a simulator, Java Real-Time System (Java RTS) 2.0 [7], for simulating real-time Java code that is compliant with the RTSJ.

To verify real-time Java code which is compliant with the RTSJ, an approach based on JPF (Java PathFinder) [8] has been proposed by Lindstrom et al. [9].

JPF is a Java model-checker which has a state-exploring JVM (Java Virtual Machine) at its core. However, the approach based on JPF to verify real-time Java code has not been implemented yet, and it only supports properties that are specified as normal Java assertions, without timing constraints.

A real-time model is a simplified representation of a real-time system. Models focus on system behaviour and abstract many details of programs [10]. More importantly, these models can be verified mechanically with real-time model-checkers. This paper investigates a modelling approach to design real-time programs written in RTSJ, by means of an industrial example. In this approach, Timed Automata [11] are used as the modelling language, since Timed Automata have well-defined mathematical properties and a simple graphical representation. Moreover, Timed Automata can capture both qualitative and quantitative features of real-time systems [12]. The next step is to mechanically verify the model using the UPPAAL model-checker [4]. UPPAAL has a graphical user interface; it is well-used and well-supported. After verifying the model, a mapping between the model features and RTSJ are used to derive the RTSJ code from the model. This approach can increase the confidence in the correctness of the program.

In this paper, we present an initial application of the proposed approach to a nontrivial example. The RTSJ code for this example is derived by hand, following the systematic approach. The current mapping we propose does not deal with timing constraints on specific time values (rather than lower- or upper-bounds) which are described in Section 3. We have also left the mapping of a number of challenging Timed Automata features for future work.

In the next section, theories and languages that have been proposed for modelling real-time systems and related work on real-time model-checking are reviewed. In Section 3, we investigate an approach to implement the behaviour exhibited by real-time models in RTSJ. A realistic industrial case study, a gear controller [13], is used as an example in this section. Section 4 provides the verification result of the gear controller and discusses the limitations of our approach, and Section 5 concludes the paper.

2 Background

Traditional formalisms for temporal reasoning deal with the qualitative aspect of time, that is, the order of certain system events (an example of a qualitative time property is: event A occurs before event B). However, real-time systems often require quantitative aspects of time. This means they need to consider the actual difference in time between certain system events.

Timed Automata [11] provide a formalism for the modelling and verification of real-time systems. Examples of other formalisms are Timed Petri Nets [14], Time Petri Nets [15], Timed Process Algebras [16], and Real-time Logics [17].

Model-checking of Timed Automata representations has become popular for the analysis of real-time systems [18]. In the last decade, there have been a number of tools developed based on Timed Automata to model and verify real-time systems, notably Kronos [3], UPPAAL [4], RT-Spin [5] and MOCHA [6]. Timed Automata can capture both qualitative and quantitative features of real-time systems [12]. For instance, liveness, fairness and nondeterminism are qualitative features and bounded response and timing delays are quantitative features that can be captured with Timed Automata. Timed Automata also have well-defined mathematical properties and a simple graphical representation.

A Timed Automaton is a finite-state automaton extended with a finite set of real-valued variables modelling clocks. Timed words in a Timed Automaton are infinite sequences in which a real-valued time of occurrence is associated with each symbol [11]. Each clock can be reset to zero with the transitions of the automaton, and keeps track of the time elapsed since the last reset. The transitions of the automaton put certain constraints on the clock values. A transition may be taken only if the current values of the clocks satisfy the associated constraints.

Figure 1 shows the Timed Automata used by the UPPAAL model checker for a clutch specified by Lindahl et al. [13]. The UPPAAL Timed Automata [19] extends Timed Automata with a number of additional features such as bounded integers, arrays and urgent locations, as discussed in Section 3.

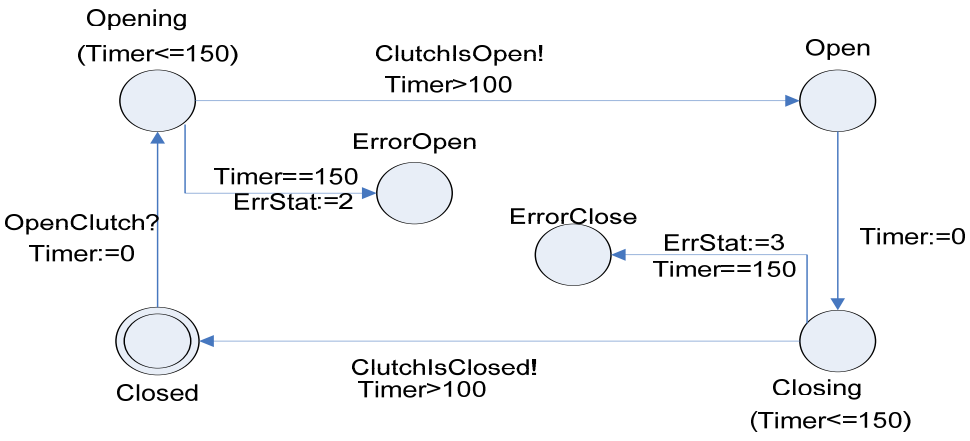


Fig. 1. Timed Automaton representing Clutch

The Clutch provides services to open or close the clutch in 100 to 150μs. In the case that opening or closing the clutch takes more than 150μs, the clutch will

stop in an error state. This example will later be used to illustrate our proposed approach.

Channels in Timed Automata are used to synchronize and communicate. For `Channel CName`, `CName?` represents receiving a message and `CName!` represents sending a message. In Figure 1, a message is sent via `OpenClutch!` and two messages are received via `ClutchIsOpen?` and `ClutchIsClosed?`. `OpenClutch?`, `ClutchIsOpen!` and `ClutchIsClosed!` are in another Timed Automaton not shown in Figure 1. `Timer` is a clock and `Timer==150` is a guard. A guard in UPPAAL is a side-effect-free statement which evaluates to a boolean. The transition from the `Opening` to the `ErrorOpen` can be taken if and only if `Timer==150` is enabled. In the `Opening` state, `Timer<=150` is an invariant. The Automaton needs to leave `Opening` before this invariant is violated.

3 From Models To Implementations

A Model is a simplified representation of the system. We use Timed Automata to describe models. These models represent the behaviour of real-time programs written in RTSJ and they can be verified mechanically with the UPPAAL model-checker. Then, we apply our approach on these models to design real-time programs which still satisfy the properties. Figure 2 shows an overview of this model-based approach, which is similar to the model-based approach proposed by Magee and Kramer to design concurrent Java programs from FSP models [10].

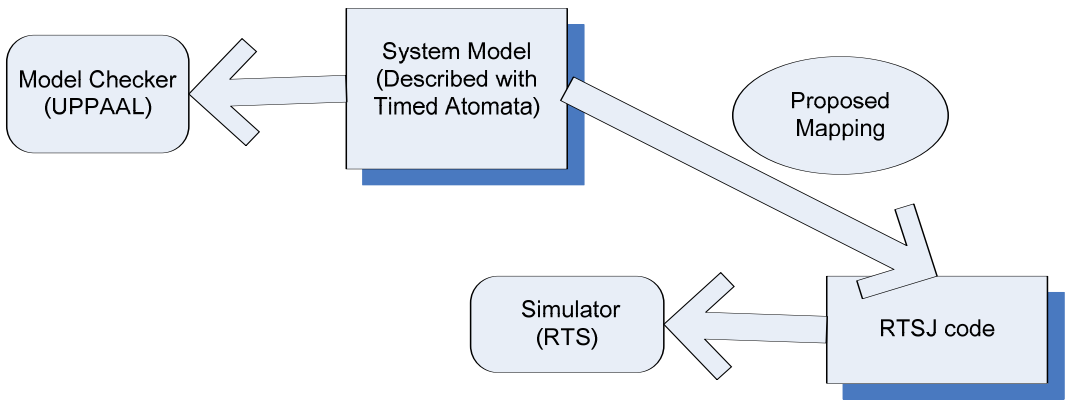


Fig. 2. Architecture

3.1 RTSJ

Real Time Specification of Java (RTSJ) [1] was introduced in January 2002. RTSJ is designed to support both hard and soft real-time applications. RTSJ adds several features to Java, such as Clocks, Time, Scoped Memory Areas which provide guarantees on allocation time, Fixed Priority Scheduling Policy, Asynchronous Events and Real-Time Threads.

Figure 3 shows part of the RTSJ code for the clutch Timed Automaton discussed in Section 2. The details of mapping the Clutch Timed Automaton to the real-time Java code are described in Section 3.3.

`Clutch` is a real-time thread. Two classes, `NonHeapRealtimeThread` and `RealtimeThread`, are defined in RTSJ to support real-time threads. Non-heap real-time threads are not targeted by the garbage collector [1].

`ClutchClock` is declared as a clock. Clocks in RTSJ are derived from an abstract class called Clock. There are three types of clocks in RTSJ:

- A "`monotonic`" clock progresses at a constant rate, suitable for timeouts.
- A "`countdown`" clock can be reset to zero, paused or continued.
- A "`CPU execution time`" clock counts the amount of time that is being consumed by a particular thread.

`MaxTimeClutch` and `CurrentTime` are a relative and absolute time respectively. In RTSJ, time is defined by three classes:

- a duration measured by a particular clock is "`relative`" time;
- "`absolute`" time is a time relative to some epoch, such as system start-up time;
- "`rational`" time is a subclass of relative time to represent the rate of certain event occurrences.

3.2 Model-based approach

An overview of the mapping for different features and expressions in UPPAAL Timed Automata is shown in Tables 1 and 2. The details of the mapping are provided in Section 3.3.

3.3 Mapping Details

Timed Automaton: Every Timed Automaton is mapped to a non-heap real-time Java thread. As non-heap real-time Java threads are not targeted by the garbage-collector, programs using such threads have no non-determinism due to garbage-collection delays or memory allocations. Each thread has a state

```
1.public class Clutch extends NonHeapRealTimeThread{
2.   public void run(){
3.        Environment env = new Environment();
4.        Clock ClutchClock = Clock.getRealtimeClock(); // Timer
5.        RelativeTime MaxTimeClutch = new RelativeTime(0,150);
6.        RelativeTime MinTimeClutch = new RelativeTime(0,100);
7.        AbsoluteTime CurrentTime = ClutchClock.getTime();
8.        String state = "Closed";
9.        while(true){
10.            if(state == "Closed"){
11.                if(env.IsReadyOpenClutch){
12.                   env.IsReadyOpenClutch = false;
13.                   env.ChannelAcknowledgeOpenClutch = true;
14.                   CurrentTime = ClutchClock.getTime();
15.                   state = "Opening";
16.                   continue;
17.                }
18.             }
19.             if(state == "Opening"){
20.                 if(((ClutchClock.getTime().subtract(CurrentTime)).compareTo(MaxTimeClutch) >= 0)){
21.                    env.ErrStat = 2;
22.                    state = "ErrorOpen";
23.                    continue;
24.                 }
25.                 if(((ClutchClock.getTime().subtract(CurrentTime)).compareTo(MinTimeClutch) > 0)){
26.                    env.ChannelAcknowledgeClutchIsOpen = false;
27.                    env.IsReadyClutchIsOpen = true;
28.                    while(!env.ChannelAcknowledgeClutchIsOpen);//busy loop
29.                    state = "Open";
30.                    continue;
31.                 }
32.             }
33.             if(state == "Open"){...}
34.             if(state == "Closing"){...}
35.       }/*while*/
36.  }/*run*/
37.}/*class*/
```

Fig. 3. Potential RTSJ code corresponding to Clutch Timed Automata

Table 1. Features Mapping Table

Feature	**Description**	**Currently supported**	**Mapped to**
Timed Automaton	a finite-state machine extended with clock variables	Yes	Real-time Thread
Broadcast channels	channels that are not blocking	Yes	A variable in the `Environment` class
Binary synchronisation	channels are declared as `chan c`	Yes	Two variables in the `Environment` class
Urgent location	time is not allowed to pass in an urgent location	Yes	Resetting the value of the Clock
Urgent synchronisation	delays must not occur if its channel is enabled	No	
Committed location	a state that cannot delay	Partially	Using RTSJ Priorities
Initialisers	used to initialise integers and arrays	Yes	Assignments in the thread constructor

Table 2. Expressions Mapping Table

Expression	Description	Currently supported	Mapped to
Assignment	an expression with a side-effect	Yes	An assignment in RTSJ
Guard	a side-effect free expression associated with a transition	Partially	An `if` condition
Invariant	a side-effect free expression associated with a state	Partially	An `if` condition except for time invariants

variable that is initialised to the initial state of the Automaton. The behaviour for the Automaton is encoded in an infinite loop in the thread `run` method. This loop contains several `if` statements on the `state` variable and each `if` statement contains the behaviour of the Timed Automaton in a state with at least one outgoing edge. As an example, Figure 3 shows the real-time thread corresponding to the Clutch Timed Automaton. Inside the `run` method of the Clutch thread in Figure 3, the string variable `state` represents the state, which is initialised to `Closed`. The infinite while loop contains four `if` statements corresponding to the four states with at least one outgoing edge: `Closed`, `Open`, `Closing` and `Opening`.

Global Variables and Broadcast Channels: To model global variables, one additional class, `Environment`, is introduced to implement the environment. The `Environment` contains global variables as static variables and all threads that need to access global variables create an instance of the `Environment` object. Broadcast Channels are considered as global variables, since they are non-blocking. For example, inside the `run` method of the Clutch thread in Figure 3, an instance of `Environment` is created to access the shared variable, `ErrStat`.

Binary synchronisation: In order to model a synchronous channel `C`, two boolean variables are introduced, `IsReadyC` and `ChannelAcknowledgeC`. The variable `IsReadyC` is set to *true* by the sender to inform the receiver that a new message is put in the channel `C` and receiver sets this boolean to *false* whenever it reads a new value from the channel variable. The `ChannelAcknowledgeC` ensures that the sender will not progress until the receiver receives the message. Whenever the sender sets its channel variable, it also sets the `ChannelAcknowledgeC` to *false* and will not continue until this variable is *true* again. Receiver sets this `ChannelAcknowledgeC` to *true* when it has read the message. The initial value of `ChannelAcknowledgeC` and `IsReadyC` are *true* and *false* respectively. In Figure 3, the clutch is the receiver for the `OpenClutch` channel. A transition from `Closed` to `Opening` is taken when a new message is put in the `OpenClutch` channel (line 11). When the clutch receives the `OpenClutch` message, it sets `IsReadyOpenClutch` to *false* to be ready for the next message and also sets `ChannelAcknowledgeOpenClutch` to *true* to inform the sender that it received the message (lines 12 and 13). On the other hand, the clutch is the sender for the `ClutchIsOpen` channel. It sets the `IsReadyClutchIsOpen` and

`ChannelAcknowledgeClutchIsOpen` variables (lines 26 and 27) and it waits until the receiver receives this message (line 28).

Urgent Locations: Time is not allowed to pass when the system is in an urgent or committed location. For a Timed Automaton, this is semantically equivalent to a location with incoming edges resetting the Timed Automaton clock and labelled with the invariant `Clock<=0`. However, interleavings with normal states are allowed. To model urgent locations we will add an assignment that saves the value of the clock after all lines of code that lead to an urgent location and then set back the clock to this value when the program leaves the code corresponding to such a location. However, the discrepancy between model and code must be noted and analysed. This feature does not occur in the gear-controller example.

Urgent Synchronisation: In an Urgent Synchronisation, if a synchronisation transition on an urgent channel is enabled, delays must not occur. In RTSJ, a priority scheduler is defined and the priority of an object that extends the Schedulable class can be set. However, even running the object with the highest priority will take some amount of time after it is enabled. The problem is even more challenging when the model contains more than one Urgent Synchronisation. We have not dealt with this feature as it did not occur in the gear-controller example.

Committed Locations: Committed Locations are urgent Locations that can not be delayed when they are enabled. Therefore, the discrepancy between model and code must be noted and analysed. This feature occurred in one Automaton, `Controller`, of our example [13]. The RTSJ code for this Automaton is available online [20].

Clocks: Each instance of a clock in a Timed Automaton is mapped to a clock in RTSJ. To check an upper- or lower-bound on a clock, a relative time is declared in Java for each bound. In addition, every thread contains an absolute time and a clock. To check the time elapsed from a particular moment, the absolute time is set to the current value of the clock. Then, the difference between the current value of the clock and the absolute time will be checked with the corresponding relative time. For example, a clock, two relative times and an absolute time are introduced for timing issues in Figure 3 (lines 4-7). In the Clutch Timed Automaton, when the transition from `Closing` to `Opening` is taken, the clock will be reset. The corresponding code for this action is shown in line 14, in which the absolute time `CurrentTime` is set to the current value of the clock `ClutchClock`. Therefore, the time elapsed from this moment will be measured. Inside the `else` statement, the program checks if the time is more than $100\mu s$ (line 25).

Guards: A transition from one state to another state can be taken if and only if the guard on the transition is enabled. A Guard is translated to an `if` statement. The code inside the `if` block corresponds to the transition updates (assignments). In Timed Automata constraints on the value of the clocks or

clock differences are only compared to integer expressions; guards over clocks are essentially conjunctions [19]. Line 20 in Figure 3 indicates that if the time elapsed since the last time at which the variable `CurrentTime` is set is equal to or more than the relative time `MaxTimeClutch`, 150, the `ErrStat` will be set to 2 and the `state` will be set to `ErrorOpen`. However, in the Timed Automaton, the guard on the transition from the `Opening` to the `ErrorOpen` is `Timer==150` and not `Timer >= 150`. Since there is no guarantee that the thread will execute this code at exactly one time, we need to be more flexible in the code than in the model. However, in this case, rather than noting and analysing the difference between model and code, we can actually modify (re-engineer) the model to match the code and then repeat the analysis of the properties we want to check on the modified model.

Dealing with non-determinism: A Timed Automaton can contain more than one transition with an enabled guard. If a state contains more than one outgoing edge with an enabled guard, one of them will be taken non-deterministically. Following the standard notion of refinement, an implementation can be more deterministic than the model. However, if we want to implement the non-determinism we can use a random variable in RTSJ. This feature occurred in one Automaton, `Interface`, of our example [13]. The RTSJ code for this Automaton is available online [20].

Invariants: The Automaton needs to leave a state before its state invariant is violated. In other words, Timed Automaton must take one of the enabled transitions if the current state invariant is violated. In RTSJ we cannot guarantee that the thread has a CPU before a certain time limit. However, we can accumulate the upper-bound of the run time of RTSJ code. To accumulate this run time upper-bound we can assign a fixed RTSJ run-time (based on the version of RTSJ and the hardware we use) to each line of code. We can then add these times to accumulate the run time of code corresponding to a state with an invariant. In Figure 1, the `Opening` has a state invariant, `Timer<=150`. Therefore, the clutch cannot stay in the `Opening` more than 150 ms and it needs to go to either `ErrOpen` or `Open` before this invariant is violated. This invariant is an assumption that should independently verified for a particular hardware and version of the RTSJ to check opening the clutch should take no more than 150 ms. In other words, executing the code in lines 14-16, 19-20 and 25 or lines 14-16 and 19-21 in Figure 3 should take less than $150\mu s$.

4 Verification result

To illustrate the applicability of the proposed method, we applied this approach to the gear-controller [13]. The model presented by Lindahl et al. contains 5 Timed Automata with a total of 63 states and 83 transitions. We recreated this

model and verified it with UPPAAL. However, the verification results were not entirely consistent with the result provided by Lindahl et al. [13]. We had to add the timing invariants on all states and increase the time bounds in the timing properties to satisfy them. The RTSJ derived from the Timed Automaton had 5 Java threads, 1320 lines of code and 16 assumptions for 16 time invariants in the model. The Timed Automata for the gear controller and the RTSJ code are both available online [20].

We unintentionally made an error in the UPPAAL model (when the clutch Automaton transitions from `Opening` to `ErrOpen`, we did not set `ErrStat` to 2). As a result, one of the system properties was violated. This property required the gear controller to notice that the clutch reached `ErrOpen`, before $300\mu s$. UPPAAL detected this error and we fixed the model. We wanted to see whether the same error would be detected in RTSJ. Therefore, we removed line 22 from Figure 3. However, the error was not detected since the offending code was not executed in the simulator as the timer never exceeds $150\mu s$. Then, we changed the invariant on `Opening` from $150\mu s$ to $50\mu s$ (line 20) and the error was detected. This shows why the model-checking approach is useful, as it detected an error in the model that is more difficult to detect in the code.

To demonstrate the discrepancy between the models, in which we make assumptions about invariants, and the code, where lines of code take a certain amount of time to execute, we changed the timing in both the model and implementation. In the original model, the invariant on the `Opening` state is $150\mu s$ and the transaction to `Open` can only be made after $100\mu s$. These properties are used to prove that if the clutch transits to `ErrOpen`, then the gear controller notices this before $300\mu s$. We changed the invariant $150\mu s$ to $2\mu s$ and the guard on transition to `Open` to $1\mu s$. In the model this is still sufficient to prove the gear controller notices the error before $4\mu s$. However, if we make these changes in the code, then the error is only detected after $50\mu s$.

5 Conclusion

In this paper, a nontrivial real-time example, a gear controller, was used to investigate a model-based approach to derive an RTSJ program from an UPPAAL model. We started from an existing UPPAAL Timed Automata for the gear controller, model checked it and followed our systematic approach to derive an RTSJ program from it. However, this approach has some limitations. As an example, when the model contains specific time values, rather than upper- or lower-bounds, it cannot be straightforwardly mapped to RTSJ code. Some other features, such as urgent synchronisation, were also left for future work.

References

1. Andy Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley and Sons Ltd, 2004.
2. Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
3. Sergio Yovine. Kronos: a verification tool for real-time systems. *Journal on Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.
4. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.
5. Stavros Tripakis and Costas Courcoubetis. Extending Promela and Spin for real time. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 329–348. Kluwer Academic Publishers, 1996.
6. Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In *Computer Aided Verification*, pages 521–525. Kluwer Academic Publishers, 1998.
7. Java SE real-time system - evaluation downloads. `http://java.sun.com/javase/technologies/realtime/rts/`. Date accessed: 20 August 2007.
8. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - a second generation of a Java model checker. In *Proceedings of the Workshop on Advances in Verification*, pages 164–169. World Scientific Publishing Company, 2000.
9. Gary Lindstrom, Peter Mehlitz, and Willem Visser. Model checking real time Java using Java PathFinder. In *3rd Int'l Symposium on Automated Technology for Verification and Analysis*, pages 444–456. Springer-Verlag, 2005.
10. Jeff Magee and Jeff Kramer. *Concurrency State Models and Java Programs*. John Wiley and Sons Ltd, 2005.
11. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets. W. Reisig and G. Rozenberg (eds.), LNCS 3098*, pages 89–90. Springer-Verlag, 2004.
12. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
13. Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller: an industrial case study using UPPAAL. Technical Report ASTEC 97/09, Advanced Software Technology, Uppsala University, 1997.
14. C. Ramchandani. Analysis of asynchronous concurrent systems by Timed Petri Nets. Technical Report TR120, MIT (Massachusetts Institute of Technology), 1974.
15. Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using Time Petri Nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, 1991.
16. Anton Wijs. Achieving discrete relative timing with untimed process algebra. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 35–46. IEEE Computer Society, 2007.
17. Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *IEEE Symposium on Foundations of Computer Science*, pages 164–169. IEEE Computer Society, 1989.
18. Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.
19. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf, 2004. Date accessed: 15 May 2007.
20. Niusha Hakimipour's home page. http://itee.uq.edu.au/∼niusha/GearControler.rar. Date accessed: 1 May 2008.

Precise Dynamic Verification of Confidentiality

Gurvan Le Guernic

INRIA-MSR - Parc Orsay Universit, 91893 Orsay - France
http://www.msr-inria.inria.fr/~gleguern/
gleguern@gmail.com

Abstract. Confidentiality is maybe the most popular security property to be formally or informally verified. Noninterference is a baseline security policy to formalize confidentiality of secret information manipulated by a program. Many static analyses have been developed for the verification of noninterference. In contrast to those static analyses, this paper considers the run-time verification of the respect of confidentiality by a single execution of a program. It proposes a dynamic noninterference analysis for sequential programs based on a combination of dynamic and static analyses. The static analysis is used to analyze some unexecuted pieces of code in order to take into account all types of flows. The static analysis is sensitive to the current program state. This sensitivity allows the overall dynamic analysis to be more precise than previous work. The soundness of the overall dynamic noninterference analysis with regard to confidentiality breaches detection and correction is proved.

1 Introduction

Language-based security is an active field of research. The majority of work on confidentiality in this field focuses on static analyses [15]. Recent years have seen a resurgence of dynamic analyses aiming at enforcing confidentiality at run time [5, 8, 16, 21]. The first reason is that nowadays it is nearly impossible for consumers to prevent the execution of "bad" code on their devices — for example, in September 2007 cybercriminals introduced malicious scripts which were executed by any browser visiting webpages of a US Consulate [9]. Moreover, there are two main potential advantages of dynamic analyses over static analyses [8]. The first one is the increased knowledge of the execution environment and behavior at run time, including the knowledge of the precise control flow followed by the current execution. This increased knowledge allows the dynamic analysis to be more precise than a static analysis in some cases; as, for example, with the program on page 93. The second advantage lies in the ability of sound information flow monitors to run some "safe" executions of an "unsafe" program while still guarantying the confidentiality of secret data. In order to take into account all indirect flows (flows originating in control statements) dynamic analyses rely on static analyses of some, but not all, unexecuted pieces of code.

This paper proposes to increase the precision of such dynamic information flow analyses. This is done by taking advantage, at the static analysis level, of the dynamic nature of the overall analysis. To do so, when statically analyzing an unexecuted piece of code, the current program state is taken into account in order to reduce the program space to analyze. The following piece of code is a

motivating example for this work. It corresponds to the body of the main loop of an Instant Messaging (IM) program. This one has the appealing "movie-like" feature of displaying messages characters by characters as they are typed.

```
1 c := getCharFromKeyboard();
2 tmp := tmp + ((int) c);
3 if ( tmp > ((int) userSecretKey) ) {
4   tmp := 0;
5   if (to = "sexyPirate") {c := specialChar}
6 };
7 send(to, c)
```

This IM program is a malware developed by "sexyPirate". When someone uses this software to communicate with a user other than sexyPirate, everything goes as expected and no secret is revealed. However, if a user communicates with sexyPirate using this IM then information about the user's secret key is leaked to the pirate. When the integer value of the characters typed by the user since the last time *tmp* has been reset to 0 reaches the integer value of the user's secret key, a special character (that sexyPirate is able to distinguish) appears on the pirate's screen. Therefore, by iterating the process, sexyPirate is able to get an accurate approximation of the user's secret key. Any sound static analysis would reject this program; and therefore, all its executions. One of the advantages of dynamic information flow analysis, if it is precise enough, is to allow use of this program for communicating with users other than sexyPirate, while still guarantying the confidentiality of the secret key in any case. However, none of the previous work are precise enough. When statically analyzing lines 4 and 5, no knowledge about the value of the variable *to* is taken into account. Therefore, the overall dynamic analysis will always consider that the value of *c* may be modified; which implies a flow from *userSecretKey* to *c*. With such dynamic analyses, line 7 must then be corrected in order to prevent any potential leakage of the value of the secret key to the outside world. In the work proposed in this paper, the static analysis used for lines 4 and 5 takes into account the run time value of the variable *to*. This allows the overall dynamic analysis proposed to detect that there is no flow from *userSecretKey* to *c* whenever *to* is different from `sexyPirate`. Therefore, it allows one to use this IM program for communicating with any user different from sexyPirate; while still preserving the confidentiality of the user's secret key even when trying to use this program to communicate with sexyPirate if the correction mechanism is applied carefully [4, 7].

The next section defines various notions used in this paper and introduces the principles of the dynamic analysis proposed in this paper. Section 3 formalizes the dynamic information flow analysis whose main properties are exposed in Sect. 4. Before presenting related works in Sect. 6, the main benefits of dynamic information flow analyses are exposed in Sect. 5. Finally, Sect. 7 concludes.

2 Definitions and Principles

A *direct flow* is a flow from the right side of an assignment to the left side. Executing "x **:=** y" creates a direct flow from y to x. An *explicit indirect flow* is a flow from the test of a conditional to the left side of an assignment in the branch executed. Executing "**if** c **then** x **:=** y **else skip end**" when c is `true` creates an explicit indirect flow from y to x. An *implicit indirect flow* is a flow from the test of a conditional to the left side of an assignment in the branch which is not executed. Executing "**if** c **then** x **:=** y **else skip end**" when c is `false` creates an implicit indirect flow from y to x.

At any execution step, a variable or expression is said to *carry variety* [2, Sect.1] if its value is not completely constrained by the public inputs of the program. In other words, a variable or expression *carries variety* if its value is influenced by the private inputs; therefore, if it may have a different value at this given execution step if the values of the private inputs were different.

A "safe" execution is a noninterfering execution. In this article, as commonly done, noninterference is defined as the absence of strong dependencies between the secret inputs of an execution and the final values of some variables which are considered to be publicly observable at the end of the execution. For every execution of a given program `P`, two sets of variable identifiers are defined. The set of variables corresponding to the secret inputs of the program is designated by $\mathcal{S}(\mathtt{P})$. The set of variables whose final value are publicly observable at the end of the execution is designated by $\mathcal{O}(\mathtt{P})$. No requirements are put on $\mathcal{S}(\mathtt{P})$ and $\mathcal{O}(\mathtt{P})$ other than requiring them to be subsets of $\mathbb{X}$ (the domain of variables). A variable x is even allowed to belong to both sets. In such a case, in order to be noninterfering, the program `P` would be required to, at least, reset the value of x. In the following definitions, we consider that a program state may contain more than just a value store. This is the reason why a distinction is done between program states (X) and value stores (σ).

Definition 1 (V-Equivalent States).
Let V be a set of variables. Two program states X_1 and X_2, containing the value stores σ_1 and σ_2 respectively, are V-equivalent *with regards to a set of variables V, written $X_1 \overset{V}{=} X_2$, if and only if the value of any variable belonging to V is the same in σ_1 and σ_2:*

$$X_1 \overset{V}{=} X_2 \quad \Longleftrightarrow \quad \forall x \in V : \sigma_1(x) = \sigma_2(x)$$

Definition 1 states a formal relation among program states. This relation defines equivalence classes of program states with regard to a given set of variables. If two program states are V-equivalent, it means that it is impossible to distinguish them solely by looking at the value of the variables belonging to the set

V. This relation is used to define the confidentiality property which is verified by the dynamic analysis presented in this paper.

Definition 2 (Noninterfering Execution).
Let $\Downarrow_s$ denote a big-step semantics. Let $\overline{\mathcal{S}(\mathtt{P})}$ be the complement of $\mathcal{S}(\mathtt{P})$ in the set $\mathbb{X}$. For all programs $\mathtt{P}$, program states X_1 and X_1', an execution with the semantics $\Downarrow_s$ of the program $\mathtt{P}$ in the initial state X_1 and yielding the final state X_1' is noninterfering, written $ni(\mathtt{P}, s, X_1)$, if and only if, for every program states X_2 and X_2' such that the execution with the semantics $\Downarrow_s$ of the program $\mathtt{P}$ in the initial state X_2 yields the final state X_2':

$$X_1 \overset{\overline{\mathcal{S}(\mathtt{P})}}{=} X_2 \;\Rightarrow\; X_1' \overset{\mathcal{O}(\mathtt{P})}{=} X_2'$$

Definition 2 states that an execution is safe — i.e. it has the desired confidentiality property — if any other execution started with the same public (non-secret) values yields a final program state which is $\mathcal{O}(\mathtt{P})$-equivalent to the final program state of the execution analyzed. It means that, by looking only at the final values of the variables observable at the end of the execution, it is impossible to distinguish this execution from any other execution whose initial program state differs only in the values of the secret inputs. Therefore, for such an execution, it is impossible to deduce information about the secret inputs of the program by looking solely at the values of the publicly observable outputs.

The dynamic analysis is based on a flow and state sensitive static analysis. During the execution, every variable is associated a tag which reflects the fact that the variable *may* or *may not* carry variety — i.e. *may* or *may not* be influenced by the secret inputs of the program. A tag store in the program state keeps track of those associations. The dynamic analysis treats directly the direct and explicit indirect flows. For implicit indirect flows, a static analysis is run on the unexecuted branch of every conditional whose test carries variety.

The static analysis is context sensitive. An unexecuted branch `P` is analyzed in the context of the program state at the time the test of the conditional, to which `P` belongs, has been evaluated. The static analysis is then aware of the exact value of the variables which do not carry variety. During the analysis, the context (value store and tag store used for the analysis) is modified to reflect loss of knowledge (in fact, only the tag store is modified). The static analysis does not compute the values of variables. Therefore, when analyzing an assignment to a variable x, the context of the static analysis is modified to reflect the fact that the static analysis does not anymore have knowledge of the precise value of the variable x. When analyzing a conditional whose test value can be computed in the current context (using only the values of the variables whose tag is $\bot$), only the branch designated by the test is analyzed. As the value of any variable which does not carry variety depends only on the public inputs, branches which are not

designated by the test value would never be executed by any execution started with the same public inputs as the monitored execution. Implicit indirect flows and explicit indirect flows must be treated with the same precision in order to prevent the creation of a new covert channel [7]. This particular point is discussed in Sect. 4. As the static analysis detects implicit indirect flows more accurately than context insensitive analyses, explicit indirect flows can also be treated more accurately.

The next section formalizes the mechanisms presented above. It presents a monitoring semantics incorporating a dynamic noninterference analysis.

3 The Monitoring Semantics

The dynamic information flow analysis and the monitoring semantics are defined together in Fig. 1. An example of the behavior of this analysis on a given execution is presented in the companion technical report [6]. Information flows are tracked using tags. At any execution step, every variable has a tag which reflects whether this variable may carry variety or not. The static analysis used for the analysis of some unexecuted branches is characterized in Fig. 2.

The language studied is an imperative language for sequential programs whose grammar follows. In this grammar, $\langle ident \rangle$ stands for a variable identifier. $\langle expr \rangle$ is an expression of values and variable identifiers. Expressions in this language are deterministic — their evaluation in a given program state always results in the same value — and are free of side effects — their evaluation has no influence on the program state.

$$
\begin{aligned}
\langle prog \rangle ::= \ & \textbf{skip} \\
| \ & \langle ident \rangle \ \textbf{:=} \ \langle expr \rangle \\
| \ & \langle prog \rangle \ ; \ \langle prog \rangle \\
| \ & \textbf{if} \ \langle expr \rangle \ \textbf{then} \ \langle prog \rangle \ \textbf{else} \ \langle prog \rangle \ \textbf{end} \\
| \ & \textbf{while} \ \langle expr \rangle \ \textbf{do} \ \langle prog \rangle \ \textbf{done}
\end{aligned}
$$

A program expressed with this language is either a skip statement (**skip**) which has no effect, an assignment of the value of an expression to a variable, a sequence of programs ($\langle prog \rangle$; $\langle prog \rangle$), a conditional executing one program — out of two — depending on the value of a given expression (`if` statements), or a loop executing repetitively a given program as long as a given expression is true (`while` statements).

3.1 A Semantics Making Use of Static Analysis Results

Let $\mathbb{X}$ be the domain of variable identifiers, $\mathbb{D}$ be the semantics domain of values, and $\mathbb{T}$ be the domain of tags. In the remainder of this article, $\mathbb{T}$ is equal to

$\{\top, \bot\}$. Those tags form a lattice such that $\bot \sqsubset \top$. $\top$ is the tag associated to variables that *may* carry variety — i.e. whose value may be influenced by the secret inputs.

The monitoring semantics described in Fig. 1 is presented as standard inference rules for sequents written in the format:

$$\zeta,\ t^{\mathrm{pc}} \ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ \zeta'$$

This reads as follows: in the monitoring execution state ζ, with a program counter tag equal to t^{pc}, program P yields the monitoring execution state ζ'. The program counter tag (t^{pc}) is a tag which reflects the security level of the information carried by the control flow. A monitoring execution state ζ is a pair $(\sigma,\ \rho)$ composed of a value store σ and a tag store ρ. A value store ($\mathbb{X} \to \mathbb{D}$) maps variable identifiers to values. A tag store ($\mathbb{X} \to \mathbb{T}$) maps variable identifiers to tags. The definitions of value store and tag store are extended to expressions. $\sigma(e)$ is the value of the expression e in a program state whose value store is σ. Similarly, $\rho(e)$ is the tag of the expression e in a program state whose tag store is ρ. $\rho(e)$ is formally defined as follows, with $FV(e)$ being the set of free variables appearing in the expression e:

$$\rho(e) = \bigsqcup_{x \in FV(e)} \rho(x)$$

The semantics rules make use of static analyses results. In Fig. 1, application of a static information flow analysis to the piece of code P in the context ζ is written: $[\![\zeta \vdash \mathtt{P}]\!]^{\sharp_{\mathcal{G}}}$. The analysis of a program P in a monitoring execution state ζ must return a subset of $\mathbb{X}$. This set, usually written $\mathfrak{X}$, is an over-approximation of the set of variables which are potentially defined in an execution of P in the context ζ. This static information flow analysis can be any such analysis that satisfies a set of formal constraints which are stated in Sect. 3.2.

The monitoring semantics rules are straightforward. As can be expected, the execution of a **skip** statement with the semantics given in Fig. 1 yields a final state equal to the initial state. The monitored execution of the assignment of the value of the expression e to the variable x yields a monitored execution state $(\sigma',\ \rho')$. The final value store (σ') is equal to the initial value store (σ) except for the variable x. The final value store maps the variable x to the value of the expression e evaluated with the initial value store ($\sigma(e)$). Similarly, the final tag store (ρ') is equal to the initial tag store (ρ) except for the variable x. The tag of x after the execution of the assignment is the least upper bound of the program counter tag (t^{pc}) and the tag of the expression computed using the initial tag store ($\rho(e)$). $\rho(e)$ corresponds to the level of the information flowing into x through direct flows. t^{pc} corresponds to the level of the information flowing into x through explicit indirect flows.

$$\frac{}{\zeta,\ t^{\mathrm{pc}}\ \vdash\ \mathbf{skip}\ \Downarrow_{\mathcal{M}}\ \zeta}$$

$$\frac{}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ x := e\ \Downarrow_{\mathcal{M}}\ (\sigma[x \mapsto \sigma(e)],\ \rho[x \mapsto \rho(e) \sqcup t^{\mathrm{pc}}])}$$

$$\frac{\zeta,\ t^{\mathrm{pc}}\ \vdash\ P^{\mathrm{h}}\ \Downarrow_{\mathcal{M}}\ \zeta^{\mathrm{h}} \qquad \zeta^{\mathrm{h}},\ t^{\mathrm{pc}}\ \vdash\ P^{\mathrm{t}}\ \Downarrow_{\mathcal{M}}\ \zeta'}{\zeta,\ t^{\mathrm{pc}}\ \vdash\ P^{\mathrm{h}}\ ;\ P^{\mathrm{t}}\ \Downarrow_{\mathcal{M}}\ \zeta'}$$

$$\frac{\rho(e) = \bot \qquad \sigma(e) = v \qquad (\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \bot\ \vdash\ P^{v}\ \Downarrow_{\mathcal{M}}\ \zeta'}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{if}\ e\ \mathbf{then}\ P^{\mathtt{true}}\ \mathbf{else}\ P^{\mathtt{false}}\ \mathbf{end}\ \Downarrow_{\mathcal{M}}\ \zeta'}$$

$$\frac{\begin{array}{c}\rho(e) = \top \qquad \sigma(e) = v \qquad (\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \top\ \vdash\ P^{v}\ \Downarrow_{\mathcal{M}}\ (\sigma^{v},\ \rho^{v}) \\ [\![(\sigma,\ \rho) \vdash P^{\neg v}]\!]^{\sharp_{\mathcal{G}}} = \mathfrak{X} \qquad \rho^{\mathrm{e}} = (\mathfrak{X} \times \{\top\}) \cup (\overline{\mathfrak{X}} \times \{\bot\})\end{array}}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{if}\ e\ \mathbf{then}\ P^{\mathtt{true}}\ \mathbf{else}\ P^{\mathtt{false}}\ \mathbf{end}\ \Downarrow_{\mathcal{M}}\ (\sigma^{v},\ \rho^{v} \sqcup \rho^{\mathrm{e}})}$$

$$\frac{\rho(e) = \bot \qquad \sigma(e) = \mathtt{false}}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ (\sigma,\ \rho)}$$

$$\frac{\rho(e) = \bot \qquad \sigma(e) = \mathtt{true}}{(\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \bot\ \vdash\ P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ \zeta'}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ \zeta'}$$

$$\frac{\rho(e) = \top \qquad \sigma(e) = \mathtt{true}}{(\sigma,\ \rho),\ t^{\mathrm{pc}} \sqcup \top\ \vdash\ P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ (\sigma',\ \rho^{\mathrm{e}})}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ (\sigma',\ \rho \sqcup \rho^{\mathrm{e}})}$$

$$\frac{\begin{array}{c}\rho(e) = \top \qquad \sigma(e) = \mathtt{false} \\ [\![(\sigma,\ \rho) \vdash P^{\mathrm{l}}\ ;\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}]\!]^{\sharp_{\mathcal{G}}} = \mathfrak{X} \\ \rho^{\mathrm{e}} = (\mathfrak{X} \times \{\top\}) \cup (\overline{\mathfrak{X}} \times \{\bot\})\end{array}}{(\sigma,\ \rho),\ t^{\mathrm{pc}}\ \vdash\ \mathbf{while}\ e\ \mathbf{do}\ P^{\mathrm{l}}\ \mathbf{done}\ \Downarrow_{\mathcal{M}}\ (\sigma,\ \rho \sqcup \rho^{\mathrm{e}})}$$

Fig. 1. Rules of the monitoring semantics

The monitored execution of a conditional whose test expression does not carry variety ($\rho(e) = \bot$) follows the same scheme as with a standard semantics. For a conditional whose test expression e carries variety, the branch (P^{v}) designated by the value of e (v) is executed and the other one ($P^{\neg v}$) is analyzed. The final value store is the one returned by the execution of P^{v}. The final tag store (ρ') is the least upper bound of the tag store returned by the execution of P^{v} and a new tag store (ρ^{e}) generated from the result of the analysis of $P^{\neg v}$ ($\mathfrak{X}$). By definition, $\rho \sqcup \rho'$ is equal to $\lambda x.\rho(x) \sqcup \rho'(x)$. The new tag store ($\rho^{e}$) reflects the implicit indirect flows between the value of the test of the conditional and the variables ($\mathfrak{X}$) which may be defined in an execution of $P^{\neg v}$. In ρ^{e}, the tag of

a variable x is equal to the initial tag of the test expression of the conditional ($\rho(e)$) if and only if x belongs to $\mathfrak{X}$; otherwise, its tag is $\bot$.

3.2 The Static Analysis Used

Fig. 2 defines some constraints characterizing a set of static analyses which can be used by the dynamic noninterference analysis. The result $\mathfrak{X}$ of a static analysis of a given program ($\mathtt{P}$) in a given context (ζ) is *acceptable* for the dynamic analysis only if the result satisfies those rules. This is written in the format: $\mathfrak{X} \models (\zeta \vdash \mathtt{P})$. In the definitions of those rules, $\{\![S^{\mathtt{true}}, S^{\mathtt{false}}]\!\}^t_v$ returns either the set $S^{\mathtt{true}}$, the set $S^{\mathtt{false}}$ or the union of both depending on the tag t and the boolean v. Its formal definition follows.

$$\{\![S^{\mathtt{true}}, S^{\mathtt{false}}]\!\}^t_v = \begin{cases} S^{\mathtt{true}} \cup S^{\mathtt{false}} & \text{iff } t = \top \\ S^v & \text{iff } t = \bot \end{cases}$$

$\emptyset \models (\ (\sigma,\ \rho) \vdash \textbf{skip}\ )$

$\{x\} \models (\ (\sigma,\ \rho) \vdash x := e\ )$

$\mathfrak{X} \models \Big(\ (\sigma,\ \rho) \vdash \mathtt{P}^{\mathrm{h}}\ ;\ \mathtt{P}^{\mathrm{t}}\ \Big)$
iff there exist $\mathfrak{X}^{\mathrm{h}}$ and $\mathfrak{X}^{\mathrm{t}}$ such that:
$\mathfrak{X}^{\mathrm{h}} \models ((\sigma,\ \rho) \vdash \mathtt{P}^{\mathrm{h}})$
let $\rho' = \rho \sqcup ((\mathfrak{X}^{\mathrm{h}} \times \top) \cup (\overline{\mathfrak{X}^{\mathrm{h}}} \times \bot))$ in $\mathfrak{X}^{\mathrm{t}} \models ((\sigma, \rho') \vdash \mathtt{P}^{\mathrm{t}})$
$\mathfrak{X} = \mathfrak{X}^{\mathrm{h}} \cup \mathfrak{X}^{\mathrm{t}}$

$\mathfrak{X} \models (\ (\sigma,\ \rho) \vdash \textbf{if}\ e\ \textbf{then}\ \mathtt{P}^{\mathtt{true}}\ \textbf{else}\ \mathtt{P}^{\mathtt{false}}\ \textbf{end}\ )$
iff there exist $\mathfrak{X}^{\mathtt{true}}$ and $\mathfrak{X}^{\mathtt{false}}$ such that:
$\mathfrak{X}^{\mathtt{true}} \models ((\sigma,\ \rho) \vdash \mathtt{P}^{\mathtt{true}})$
$\mathfrak{X}^{\mathtt{false}} \models ((\sigma,\ \rho) \vdash \mathtt{P}^{\mathtt{false}})$
$\mathfrak{X} = \{\![\mathfrak{X}^{\mathtt{true}}, \mathfrak{X}^{\mathtt{false}}]\!\}^{\rho(e)}_{\sigma(e)}$

$\mathfrak{X} \models \Big(\ (\sigma,\ \rho) \vdash \textbf{while}\ e\ \textbf{do}\ \mathtt{P}^{\mathrm{l}}\ \textbf{done}\ \Big)$
iff there exists $\mathfrak{X}^{\mathrm{l}}$ such that:
let $\rho' = \rho \sqcup ((\mathfrak{X}^{\mathrm{l}} \times \top) \cup (\overline{\mathfrak{X}^{\mathrm{l}}} \times \bot))$ in $\mathfrak{X}^{\mathrm{l}} \models ((\sigma, \rho') \vdash \mathtt{P}^{\mathrm{l}})$
$\mathfrak{X} = \{\![\mathfrak{X}^{\mathrm{l}}, \emptyset]\!\}^{\rho(e)}_{\sigma(e)}$

Fig. 2. Constraints on the static analysis results

4 Properties of the Monitoring Semantics

Section 3 formally defines the dynamic information flow analysis proposed in this article. In the current section, this dynamic noninterference analysis is proved to be sound with regard to the enforcement of the notion of noninterfering execution given in Definition 2. This means that, any monitor enforcing noninterference using this dynamic analysis would be able to ensure that: for any two monitored executions of a given program P started with the same public inputs (variables which do not belong to $\mathcal{S}(\mathtt{P})$), the final values of observable outputs (variables which belong to $\mathcal{O}(\mathtt{P})$) are the same for both executions.

Theorem 1 proves that the dynamic analysis is sound with regard to information flow *detection*. Any variable, whose tag at the end of the execution is $\bot$, has the same final value for any executions started with the same public inputs.

Theorem 1 (Detection Soundness).
For all programs $\mathtt{P}$ *and states* $(\sigma_1,\ \rho_1)$, $(\sigma'_1,\ \rho'_1)$, $(\sigma_2,\ \rho_2)$ *and* $(\sigma'_2,\ \rho'_2)$*:*

$$\left.\begin{array}{l}(\sigma_1,\ \rho_1),\ \bot\ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ (\sigma'_1,\ \rho'_1)\\ (\sigma_2,\ \rho_2),\ \bot\ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ (\sigma'_2,\ \rho'_2)\\ \forall x \notin \mathcal{S}(\mathtt{P}).\ \sigma_1(x) = \sigma_2(x)\\ \forall x \in \mathcal{S}(\mathtt{P}).\ \rho_1(x) = \top\end{array}\right\} \Rightarrow\ \Big(\forall x.\ (\rho'_1(x) = \bot)\ \Rightarrow\ (\sigma'_1(x) = \sigma'_2(x))\Big)$$

Proof (Proof summary). The detailed formal proof can be found in the companion technical report [6]. The proof aims at showing that the invariant *which relates the fact, for every variable, of having a* $\bot$ *tag and not carrying variety — i.e. not being influenced by the secret inputs — is preserved during the execution. The invariant preservation is obvious for* **skip** *statements. If the tag of a variable* x *after an assignment of* e *to* x *is* $\bot$*, then it means, first, that the control flow does not carry variety* ($t^{pc} = \bot$) *and therefore that, with similar public inputs, the assignment will always be executed. It also means that the expression does not carry variety and the invariant property is preserved by the execution of assignments. For sequence statements, if the invariant is preserved by the first and second statements then it is preserved by the sequential execution of both statements. If the tag of the condition of a branching statement is* $\bot$*, then any execution started with the same public inputs evaluates the same branch as the execution monitored. As, by induction, the execution of the branch preserves the invariant, the invariant is also preserved. If the condition's tag is* $\top$*, as the tag of the control flow* (t^{pc}) *is updated to* $\top$*, all the variables assigned to in the branch have a tag of* $\top$*. Additionally, any variable which may have been assigned to in the other branch are in the set returned by the static analysis, therefore their tag also becomes*$\top$*. Hence, the invariant relating* $\bot$ *and not carrying variety is preserved by the execution of a branching statement. If the tag of the condition of a loop is* $\bot$ *then, if its value is false the loop is equivalent to a* **skip** *and the*

invariant is preserved. If the tag of a true condition is $\bot$, the proof follows by induction. If the condition's tag is $\top$ then, for the same reasons as the case for branching statements, the tag of all the variables whose value may be modified by the loop becomes $\top$. Therefore, the invariant property is preserved by loops.

More informally, theorem 1 compares any 2 executions which are such that: any public inputs ($x \notin \mathcal{S}(\mathtt{P})$) have the same initial value, and any secret input have an initial tag of $\top$ in order to let the dynamic analysis know that their content is secret. Theorem 1 states that if the final tag of a variable is $\bot$ — therefore, that the analysis considers its value to be safely accessible — then the final values of this variable for both executions — any 2 executions which have the same public inputs — are the same. Therefore, an attacker, who looks at the final value of a variable whose final tag is $\bot$, sees the same value for all the executions which have different secret inputs but the same public inputs. Hence, an attacker does not learn anything about the secret inputs by observing the final values of variables whose final tag is $\bot$. Therefore, to sanitize the final state of an execution, it is sufficient to "securely" (as will be explained in the following discussion) reset the value of observable variables ($\mathcal{O}(\mathtt{P})$) whose final tag is $\top$.

As shown by Le Guernic and Jensen [7], if the correction of "bad" information flows is done without enough care, the correction mechanism itself can become a new covert channel carrying secret information. Indeed, theorem 1 states that the final value of a variable, whose final tag is $\bot$, is not influenced by the value of the secret inputs. However, if the final tag of a variable is influenced by the values of the secret inputs, then it means that for some secret input values the final tag of this variable will be $\bot$ and for other secret input values it will be $\top$. Hence, it means that, for some secret input values, the correction mechanism will reset the final value of the variable; and for other secret input values, the final value of the variable will not be reset. Therefore, by checking if the final value of this variable has been reset or not, an attacker can learn information about the secret input values. Theorem 2 proves that, for the analysis presented in this paper, the final tag of a variable does not depend on the secret inputs of the program. Therefore, any variable belonging to $\mathcal{O}(\mathtt{P})$ whose final tag is not $\bot$ can safely be reset to a default value without creating a new covert channel.

Theorem 2 (Correction Soundness).
For all programs $\mathtt{P}$ and states $(\sigma_1,\ \rho_1)$, $(\sigma'_1,\ \rho'_1)$, $(\sigma_2,\ \rho_2)$ and $(\sigma'_2,\ \rho'_2)$:

$$\left.\begin{array}{l}(\sigma_1,\ \rho_1),\ \bot\ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ (\sigma'_1,\ \rho'_1)\\ (\sigma_2,\ \rho_2),\ \bot\ \vdash\ \mathtt{P}\ \Downarrow_{\mathcal{M}}\ (\sigma'_2,\ \rho'_2)\\ \forall x \notin \mathcal{S}(\mathtt{P}).\ \sigma_1(x) = \sigma_2(x)\\ \forall x \in \mathcal{S}(\mathtt{P}).\ \rho_1(x) = \top\\ \rho_1 = \rho_2\end{array}\right\} \Rightarrow\ \left(\rho'_1 = \rho'_2\right)$$

Proof (Proof summary). The detailed formal proof can be found in the companion technical report [6]. In order to be able to use induction, a generalization of the theorem stated above is proved with two differences in its statement: program counters (t^{pc}) must only be the same for the two executions and variables whose tag is $\bot$ must have the same value in both executions instead of the hypotheses based on $\mathcal{S}(P)$. The case for **skip** *is direct. For assignments, the only tag modified is the one of the variable assigned. As the two tag stores are initially equal, $\rho_1(e)$ is equal to $\rho_2(e)$. Therefore, both tag stores are equal after the execution of the assignment. The case for sequences goes by induction. For* `if` *statements, if the same branch is executed then by induction the theorem holds. If two different branches are executed then the tag of the program counter is $\top$. Therefore, the tag of every assigned variable becomes $\top$. Additionally, the unexecuted branch is analyzed and the tag of every variable which may have been assigned to is set to $\top$. Fig. 2 constrains the analysis to make the same choices as the execution with regard to which subbranches to ignore and which ones to analyze. Therefore, the set of variables returned by the analysis of the unexecuted branch is exactly the set of variables whose tag would have been set to $\top$ by an execution of this branch. Therefore, whatever branch is executed or analyzed, the same set of variables have their tag set to $\top$. Hence, the final tag stores are equal after the execution of the* `if` *statement. For* `while` *statements, if the condition evaluates to the same value then the inductive hypothesis implies that the theorem holds. Otherwise, it means that its tag is $\top$. Therefore the final tag store is the least upper bound of the initial tag store and a new tag store ρ^e. This new tag store is either the tag store returned by the execution of the* `while` *statement (executing the body at least once) with program counter tag (t^{pc}) equal to $\top$ or the analysis of the same statement. As for* `if` *statements, in both cases the tags of the exact same set of variables are set to $\top$. Hence, the theorem holds.*

5 Benefits of Monitoring Compared to Static Analyses

Monitoring an execution has a cost. So, what are the main benefits of noninterference monitoring compared to static analyses? The first concerns the possibility that a monitoring mechanism can be used to change the security policy for each execution. In the majority of cases, running a static analysis before every execution would be more costly than using a monitor. The second reason is that noninterference is a rather strong property. Many programs are rejected by static analyses of noninterference. In such cases it is still possible to use a monitoring mechanism with the possibility that some executions will be altered by the monitoring mechanism. However behavior alteration is an intrinsic feature of any monitoring mechanism. Monitoring noninterference ensures confidentiality while still allowing testing with regard to other specifications using unmonitored executions as perfect oracle — at least as perfect as the original program.

There are two main reasons why it is interesting to use a noninterference monitor on a program rejected by a static analysis. The first one is that a monitoring mechanism may be more precise than static analyses because during execution the monitoring mechanism gets some accurate information about the "path behavior" of the program. As an example, let us consider the following program where *h* is the only secret input and *l* the only other input (a public one).

```
1 if ( test1(l) ) then tmp := h else skip end;
2 if ( test2(l) ) then x := tmp else skip end;
3 output x
```

Without information on *test1* and *test2* (and often, even with), a static analysis would conclude that this program is unsafe because the secret input information could be carried to *x* through *tmp* and then to the output. However, if *test1* and *test2* are such that no value of *l* makes both predicates true, then any execution of the program is perfectly safe. In that case, the monitoring mechanism would allow any execution of this program. The reason is that, *l* being a public input, only executions following the same path as the current execution are taken care of by the monitoring mechanism. So, for such configurations where the branching conditions are not influenced by the secret inputs, a monitoring mechanism is at least as precise as any static analysis — and often more precise.

The second reason lies in the granularity of the noninterference property. Static analyses have to take into consideration all possible executions of the program analyzed. This implies that if a single execution is unsafe then the program (thus all its executions) is rejected. Whereas, even if some executions of a program are unsafe, a monitor still allows this program to be used. The unsafe executions, which are not useful, are altered to enforce confidentiality while the safe executions are still usable. For example, the program on page 83 being interfering, any static noninterfering analysis rejects this program. Therefore, users would be advised not to use this program at all. However, using a noninterference monitor, it is possible to safely use this program. When communicating with any user other than sexyPirate, monitored executions of this program have their normal behavior. When communicating with sexyPirate, monitored executions are safely detected as potentially interfering and can therefore be corrected to prevent any secret leakage. Of course, when attempting to communicate with sexyPirate, executions of this program are altered and it is therefore not possible to communicate with sexyPirate. However, this is the desired behavior of a noninterference monitor when confidentiality is more important than the service provided by the program.

6 Related Work

The vast majority of research on noninterference concerns static analyses and involves type systems [13, 15]. Some "real size" languages together with security type system have been developed (for example, JFlow/JIF [11] and FlowCaml [14]).

Dynamic information flow analyses [1, 3, 19, 20] are not as popular as static analyses for information flow, but there has been interesting research. For example, RIFLE [17] is a complete run-time information flow security system based on an architectural framework and a binary translator. Masri et al. [10] present a dynamic information flow analysis for structured or unstructured languages. Venkatakrishnan et al. [18] propose a program transformation for a simple deterministic procedural language that ensures a sound detection of information flows. Trishul [12] is an interesting implementation of a Java Virtual Machine integrating a dynamic information flow control mechanism. Yoshihama et al. [21] propose a Java Architecture for Web Applications with "direct" and "explicit indirect" information flow control abilities (as they acknowledge, they do not handle "implicit indirect" flows). One of the main interest of those works is the size of the language addressed. However, none of those five later works prove that the correction mechanisms of "bad" flows proposed do not create a new covert channel that can reveal secret information — see, e.g., [7] — or even, for some of them, that the detection mechanism is sound with regard to their notion of information flow. In fact, those analyses and correction mechanisms are likely to create a new covert channel. Theorem 2 proves that a correction mechanism of "bad" flows can be based on the dynamic analysis proposed in this paper as the results of the dynamic analysis are the same for every executions started with the same public inputs. More recently, Shroff, Smith, and Thober [16] proposed a dynamic information flow analysis which tracks direct flows and collects indirect flows dynamically. The information collected about indirect flows is transferred from one execution to another using a cache mechanism. After an undetermined number of executions, the analysis will know about all indirect flows in the program and thus will then be sound with regard to the detection of all information flows. This information about indirect flows can be precomputed using a static analysis at the cost of a decrease of precision. Using this approach they are able to handle a language including alias and method calls.

Contrary to common assumption, none of the related works on dynamic information flow analysis known to the author take enough context information into account to detect that the program on page 83 is noninterfering when using it to communicate with users other than sexyPirate. For example, the transformation of Venkatakrishnan et al. [18] updates the security label of c with the security label of the condition on line 3 before executing line 7. Therefore, at line 7, c is always considered as secret even if the user is not communicating

with sexyPirate. When executing the assignment of line 5, the program counter of Shroff et al.'s work [16] contains a reference to the program point of line 3 and therefore is added to the set of source of implicit flows to c. Consequently, any complete implicit dependency cache contains a reference to the implicit flow from line 3 to line 7. As the test of line 3 is always executed, Shroff et al.'s work always considers line 7 as displaying secret information. The dynamic analysis proposed in this paper *is* able to detect the noninterfering behavior of the program on page 83 when communicating with someone other than sexyPirate.

7 Conclusion

This article addresses the problem of information flow verification and correction at run time in order to enforce the confidentiality of secret data. The confidentiality property to monitor is expressed using the property of noninterference between secret inputs of the execution and its public outputs. The language taken into consideration is a sequential language with assignments and conditionals (including loops). The main difference between the monitoring mechanism proposed in this article and the ones of related works lies in the static analysis used to detect implicit indirect flows. The static information flow analyses used by the dynamic analysis proposed in this article are sensitive to the current program state. This allows the overall dynamic information flow analysis to increase the precision of the detection of implicit and explicit indirect flows. In Sect. 4, the proposed noninterference monitor is proved to be sound both with regard to the detection of information flows and with regard to their correction when necessary.

Acknowledgments: The author is grateful to Anindya Banerjee, Gérard Boudol, Thomas Jensen, Andreï Sabelfeld and David Schmidt for their helpful feedback on an earlier version of this work.

Bibliography

[1] J. Brown and T. F. Knight, Jr. A minimal trusted computing base for dynamically ensuring secure information flow. Technical Report ARIES-TM-015, MIT, 2001.

[2] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[3] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[4] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[5] G. Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proc. Computer Security Foundations Symp.*. IEEE, 2007.

[6] G. Le Guernic. Precise dynamic verification of noninterference. Technical report, INRIA, July 2008. http://hal.inria.fr/inria-00162609/fr/.

[7] G. Le Guernic and T. Jensen. Monitoring Information Flow. In *Proc. W. on Foundations of Computer Security*, pages 19–30. DePaul University, 2005.

[8] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based Confidentiality Monitoring. In *Proc. Annual Asian Computing Science Conf.*, volume 4435 of *LNCS*, 2006.

[9] J. Leyden. Trojan planted on US Consulate website. The Register, Sept. 2007.

[10] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *Proc. Int. Symp. on Software Reliability Engineering*, pages 198–209. IEEE, 2004.

[11] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. Symp. Principles of Programming Languages*, pages 228–241, 1999.

[12] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. In *Proc. W. on Run Time Enforcement for Mobile and Distributed Systems*, volume 197, pages 3–16, 2007. Elsevier.

[13] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. Int. Conf. on Functional Programming*, pages 46–57. ACM Press, 2000.

[14] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. on Programming Languages and Systems*, 25(1):117–158, 2003.

[15] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, 2003.

[16] P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. Computer Security Foundations Symp.*. IEEE, 2007.

[17] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proc. Int. Symp. on Microarchitecture*, 2004.

[18] V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Proc. Int. Conf. on Information and Communications Security*, volume 4307 of *LNCS*, pages 332–351. Springer-Verlag, 2006.

[19] C. Weissman. Security controls in the adept-50 timesharing system. In *Proc. AFIPS Fall Joint Computer Conf.*, volume 35, pages 119–133, 1969.

[20] J. P. L. Woodward. Exploiting the dual nature of sensitivity labels. In *Proc. Symp. Security and Privacy*, pages 23–31, 1987.

[21] S. Yoshihama, T. Yoshizawa, Y. Watanabe, M. Kudoh, and K. Oyanagi. Dynamic information flow control architecture for web applications. In *Proc. European Symp. on Research in Computer Security*, volume 4734 of *LNCS*, pages 267–282. Springer-Verlag, 2007.

Author Index