Deductive Verification of a Hypervisor Model

Vlad Rusu, Gilles Grimaud, Michaël Hauspie, François Serman

Deductive Verification of a Hypervisor Model

Vlad Rusu[1], Gilles Grimaud[2], Michaël Hauspie[2], and François Serman[2]
[1] Inria
Vlad.Rusu@inria.fr
[2] Université de Lille
{Gilles.Grimaud|Michael.Hauspie|Francois.Serman}@univ-lille.fr

Abstract. We propose a deductive-verification approach for proving partial-correctness and invariance properties on arbitrary transition systems, and demonstrate the approach on a security hypervisor model. Regarding partial correctness, we generalise the recently-introduced formalism of Reachability Logic, currently used as a language-parametric program logic, to arbitrary transition systems. We propose a sound and relatively-complete proof system for the resulting logic. The soundness of the proof system is formally established in the Coq proof assistant, and the mechanised proof provides us with a generic Reachability-Logic prover within Coq for transition-system specifications. The relative completeness of the proof system, although theoretical in nature, also has a practical value, as it induces a proof strategy that is guaranteed to prove all valid formulas on a given transition system. The strategy reduces partial-correctness verification to invariant verification; for the latter we propose an incremental technique in order to deal with the case-explosion problem. All these various techniques were instrumental in enabling us to deal with a nontrivial case study: proving that a Coq model of a security hypervisor meets its expected requirements, expressed as invariants and partial-correctness properties, within reasonable time and effort limits. We also report on some experiments with a C+ARM assembly implementation of our hypervisor in order to confirm the fact that it introduces a limited amount of execution-time overhead to operating-system calls.

1 Introduction

Partial-correctness and invariance properties are among the most important properties of algorithmic programs. Partial correctness can be broadly stated as: on all terminating executions, a given relation holds between a program's initial and final states; and invariants are state predicates that hold in all states reachable from a given set of initial states. Such properties have been formalised in several program logics and are at the heart of many program-verification tools.

In this paper we propose to generalise the verification of partial-correctness and invariance properties, from programs, to arbitrary transition systems. The motivation is that program verification, although desirable since it ensures a high degree of trust in the verification's results, is not always feasible. For example, our case study in this paper: a security hypervisor for ARM machine code, was implemented in a combination of C and ARM assembler; formally verifying the

implementation would first require us to formally define the semantics of this language combination: presumably, a huge task, with little potential for reuse.

How can one specify such properties on transition systems, and how can one verify them? One possibility must be ruled out from the start: Hoare logics [1]. Although specifically designed to specify partial correctness and invariants, Hoare logics intrinsically require *programs*, as their deduction rules focus on how instructions modify state predicates; and we do not target programs but more abstract models - transition systems. It is worth noting that transition systems are a well-known formalism, which can be used to specify broad classes of systems.

Regarding partial-correctness and invariance properties, one could express them in temporal logic [2] and use a model checker to prove the resulting temporal-logic formulas. However, model checkers are limited to *essentially* finite-state systems (perhaps up to some state abstraction), a limitation we want to avoid.

Contribution. We thus propose a generic approach implemented in the Coq proof assistant [3], whose expressive logic allows one to encode arbitrary transition systems. We express partial-correctness properties by generalising *Reachability Logic* [4–7, 9] (hereafter, RL) to arbitrary transition systems: RL, a language-parametric program logic, is itself a generalisation of Hoare logics, designed to loosen the syntactical dependency between a Hoare logic and the programming language it is built upon. We also define *Abstract Symbolic Execution* (hereafter, ASE), a generalisation of symbolic execution from programs to arbitrary transition systems, and propose a deductive system for RL in this setting. We prove the soundness of the proof system both on paper and in the Coq proof assistant.

The Coq soundness proof provides us with a Coq-certified RL proof system for transition-system specifications. We also prove a relative-completeness result for our proof system, which, although theoretical in nature, also has a practical value, since it amounts to a strategy for applying the proof system, which does succeed on all valid RL formulas on a given transition system. The strategy reduces partial-correctness verification to invariant verification; for the latter we employ a standard invariant-strengthening technique that amounts to strengthen a state predicate until it becomes *inductive*, i.e., stable under the transition relation. A known problem affecting invariant strengthening is *case explosion*: a tentative invariant consisting of n conjuncts, to be proved stable over a transition relation symbolically defined by m cases, generates $n * m$ typically large goals to prove, which can become overwhelming since both m and n are typically large for nontrivial systems. The actual situation is even worse, since the conjuncts of an inductive invariant are, typically, only discovered incrementally by users (by analysing why a tentative invariant fails to be preserved by some transitions). Moreover, when new conjuncts are added to a given tentative invariant, the user has to re-attempt proving the stability of its whole set of conjuncts, new *and* old, which can be, in our experience, a major source of frustration. We thus propose a incremental technique to mitigate the need for such costly proof repetition.

All these techniques (proof system, strategy reducing partial correctness to invariants, incremental invariant strengthening) were instrumental in enabling us to verify a nontrivial case study with a reasonable amount of time and effort.

The case study itself is a transition-system model of a security hypervisor for ARM machine code that we designed [10]. The hypervisor alternates between a simple static code analysis/instrumentation and dynamic code execution after the analysis/instrumentation has deemed a given code section secure. It is designed to minimise the execution-time overhead induced by time-costly alternations between the analysis/instrumentation and execution phases. We formally prove that the hypervisor fulfills its expected functional requirements: it hypervises all "dangerous" instructions in any given piece of code while not semantically altering the code in question. We also report on experiments with a C+ARM assembly implementation of the hypervisor, which confirms the fact that hypervision introduces little execution-time overhead when applied to operating-system calls.

Related Work. We give related work regarding Reachability Logic, symbolic execution, formal verification in the Coq proof assistant, and security hypervisors.

Reachability Logic [4–7, 9] is a formalism initially designed for expressing the operational semantics of programming languages and for specifying programs in the languages in question. Languages whose operational semantics is specified in (an implementation of) RL include those defined in the $\mathbb{K}$ framework [11], e.g., Java and C. Once a language is formally defined in this manner, partial-correctness properties of programs in the language can be formally specified using RL formulas. The verification is then performed by means of a sound deductive system, which is also complete relative to an oracle deciding certain first-order theories. Recently, it has been noted that RL can be generalised to other classes of systems, i.e., rewriting-logic specifications [12, 13]. In this paper we generalise RL and its proof system to a broader class of specifications - transition systems.

One significant difference between the proposed proof system and earlier ones is that our completeness result is not only a theoretical one but has practical applications: it gives a strategy for applying the proof system's rules that reduces the verification of RL formulas to that of inductive predicates, for which a systematic proof technique (the user strengthens a given predicate with new conjuncts obtained by analysing why the predicate fails to be inductive) exists.

The papers [4–7, 9] describe several variants of RL (earlier known as *matching logic*[3]). The version of RL that we are here generalising is the *all-paths* version, which can deal with nondeterministic programs, in contrast with the *one-path* version, limited to deterministic ones. We note that Coq soundness proofs have also been achieved for proof systems of various versions of RL [6–8]. Those proofs did not grow into practically usable RL interactive provers, however, because the resulting Coq frameworks require too much work in order to be instantiated even on the simplest programming languages[4]. By contrast, our ambition is to obtain a practically usable, interactive certified RL prover within Coq, for transition-system specifications and applicable to nontrivial case studies.

[3] Matching logic now designates a logic for reasoning on program configurations [14]. These changes in terminology are a side effect of the field being relatively recent.

[4] To our best understanding, obtaining certified RL interactive provers was not the goal of our colleagues; rather, they implemented automatic, non-certified provers.

Our approach is based on a generalisation of symbolic execution, an old technique that consists in executing programs with symbolic values rather than concrete ones [15]. Symbolic execution has more recently received renewed interest due to its applications in program analysis, testing, and verification [16–21]. Symbolic execution-like techniques have also been developped for rewriting logic, including rewriting modulo SMT [22] and narrowing-based analyses [23, 24].

Formal verification in Coq is a vast field; we here give some relevant references regarding program verification as well as the verification of higher-level specifications. An early example of a significant system verified in Coq is a distributed reference counter for, e.g., garbage collection, specified in an algorithmic style [25]. More recently, separation logic has been used to verify Coq implementations of garbage collectors [26]. These two references illustrate a general trend that goes from the early verifications of algorithms/specifications to the more recent verifications of programs/implementations. Of course, verifying programs is a desirable goal, but, as discussed earlier in this paper, it is not always feasible.

Major programming languages such as Java and C are the object of formalisations using Coq. We mention the Krakatoa [27] toolset for JAVA, which together with its counterpart Frama-C [28] for C are front-ends to the Why tool [29], which generates proof obligations to be discharged in Coq (among other back-end provers). In the area of C-related works in Coq an essential reference is CompCert [30], a project that, however, does not aim at program verification but at developing a trusted compilation chain for C. A Coq verifier for a low-level extensible programming language is Bedrock [31], and a language-parametric program-verification approach in Coq is presented in [35]. The current program-verification and more generally programming language-related research in Coq is active, as illustrated, e.g., by the *Coq for Programming Languages* workshop series [32] affiliated with the conference POPL. This research is becoming increasingly accessible to beginners thanks to freely available online books [33, 34]. Major international research projects around Coq, such as DeepSpec [36] aim at specifying and verifying full functional correctness of software and hardware.

A hypervisor is for an operating system what an operating system is for a process: it creates a virtual version of the underlying hardware. A guest operating system running on top of such a hypervisor "believes" it is the only one operating on the hardware, whereas in reality there can be several such guests. Virtualisation ensures that guests are mutually isolated, and, as a consequence, it guarantees the confidentiality and integrity of their respective data. It also ensures security properties for the hypervisor itself and for the underlying hardware by preventing any unauthorized access of guests to the hardware. Two kinds of virtualisation can be distinguished: para-virtualisation and full virtualisation.

Para-virtualisation prevents privileged operations (e.g., updating memory-management data structures) to be directly executed by guest operating systems, by "wrapping" them into calls to hypervisor primitives. Thus, para-virtualisation modifies the source code of its guests. It can be viewed as a collaboration between guests and hypervisor. The Xen [37] hypervisor is an example of this category. By contrast, full virtualisation does not require a guest's code to be modified;

rather, the hypervisor implements mechanisms to take back control when guests execute privileged instructions. Guest operating systems are thus typically run with a lower level of privilege; they trigger exceptions when attempting to run privileged instructions, which are then handled by the hypervisor. VMWare [38] and Qemu [39] are examples of this category. Our hypervisor [10], used as use case study for this paper, is also based on of full virtualisation.

Finally, abstract interpretation [40] provides us with a useful terminology (abstract and concrete states, abstract and concrete executions, etc) which we found convenient for defining abstract symbolic execution, based on which our proof system for RL is built and its soundness/completeness results are proved.

Paper Organisation. In Section 2 we introduce abstract symbolic execution, a extension of symbolic execution from programs to arbitrary transition systems. In Section 3 we generalise Reachability Logic to transition systems and introduce our proof system, its soundness and completeness results, and the corresponding Coq formalisation. Section 4 presents the Coq proof of soundness for the proof system and the application of the resulting Coq RL prover to our hypervisor case study. We also report on experiments with an implementation of the hypervisor, which confirm the fact that hypervision does not introduce too much execution-time overhead when applied to its natural targte: operating-system calls. Section 5 concludes and discusses future work. For better readability the proofs of most technical results are moved in a separate Appendix. The Coq and the C+ARM codes related to this paper are available at `https://project.inria.fr/rlase`.

2 Abstract Symbolic Execution

We borrow terminology from abstract interpretation in order to define a notion of abstract symbolic execution (ASE) for transition systems, over which our proof system is built. We thus assume some basic knowledge of abstract interpretation [40], including the following standard abstract-interpretation constructs:

- A set of concrete states S and a concrete transition relation $\rightarrow \subseteq S \times S$.
- A set of abstract states $S^\#$ organised as a lattice$(\sqsubseteq, \bot, \top, \vee, \wedge)$ where $\sqsubseteq \subseteq S^\# \times S^\#$ is a partial order; $\bot$ and $\top$ are minimal, respectively, maximal elements with respect to the order; $\vee : S^\# \times S^\# \to S^\#$ is the *join* operation, satisfying $q, q' \sqsubseteq q \vee q'$ for all abstract states $q, q' \in S^\#$; and $\wedge : S^\# \times S^\# \to S^\#$ is the *meet* operation, satisfying $q \wedge q' \sqsubseteq q, q'$ for all $q, q' \in S^\#$.
- An abstract transition function $\xrightarrow{\#}: S^\# \to S^\#$. We use a function (rather than a relation) because it is technically more convenient and does not restrict generality: the nondeterminism and partiality of a relation are expressed by a function returning a join of abstract states resp. the bottom element $\bot$.
- A concretisation function $\gamma : S^\# \to P(S)$, where $P(S)$ is the powerset of S;
- Abstract transitions simulate concrete ones: for all $s, s' \in S$, $q \in S^\#$, if $s \rightarrow s'$ and $s \in \gamma(q)$ then $s' \in \gamma(\xrightarrow{\#}(q'))$, cf. commuting diagram in Figure 1 (left).

Other important constructions of abstract interpretation (abstraction functions, Galois connections, widening, ...) are not required here; our goal is just to use the terminology of abstract interpretation in order to conveniently define abstract symbolic execution, without redefining everything from the ground up.

Fig. 1. (left) abstract simulate concrete (right) concrete backwards-simulate abstract.

In addition to the above standard constructions we have the following additional assumptions in ASE: the concretisation function γ completely defines the order $\sqsubseteq$ and it "distributes" over $\vee$ and $\wedge$; each abstract state has a complement, consistent with the concretisation function; and the inverse of the concrete transition relation simulates the inverse of the abstract transition function.

Assumption 1 (Additional assumptions for ASE 1) *For all abstract states q, q', $q \sqsubseteq q'$ iff $\gamma(q) \subseteq \gamma(q')$; $\gamma(q \vee q') = \gamma(q) \cup \gamma(q')$; $\gamma(q \wedge q') = \gamma(q) \cap \gamma(q')$; and for each abstract state q there is an abstract state $\overline{q}$ such that $\gamma(\overline{q}) = S \setminus \gamma(q)$.*

Assumption 2 (Additional assumptions for ASE 2) *For all abstract states q and all concrete state $s' \in \gamma(\overset{\#}{\rightarrow}(q))$, there exists $s \in \gamma(q)$ such that $s \rightarrow s'$.*

The latter is graphically illustrated by the commuting diagram in Fig. 1 (right).

ASE in Coq. The above concepts and assumptions enable us in forthcoming sections to define and to reason about our proof system for RL "on paper". When we translate them to Coq we use the following representation[5]. *Concrete states* have an arbitrary Coq type `State`. *Concrete transitions* are predicates on pairs of states; in Coq this is written in the so-called *curried form* as `trans: State → State → Prop`, where `Prop` is Coq's predefined type for logical statements and $\rightarrow$ denotes both the "functional arrow" and logical implication (which, in Coq's logic, are essentially the same thing). *Abstract states* are defined to be the type of predicates on states: `AbsState : Type := State → Prop`.

The concretisation function is expressed as a relation between abstract and concrete states: `gamma(q:AbsState)(s:State) := q s`, i.e., the concretisation of a abstract state `q` contains the concrete states `s` than satisfy `q`. Then, the order, meet, join, bottom, top, and complement constructions on abstract states are defined using, respectively, Coq's predefined implication, disjunction, conjunction, `False`, `True`, and negation constants and operations on type `Prop`. For example, the order relation is `ord(q q': AbsState) := ∀ s, q s → q' s`. Finally, the *abstract transition function* is defined by a form of *strongest postcondition*: `absTran(q:AbsState):AbsState:= fun s⇒ ∃s,q s'∧tran s' s`.

The ASE assumptions shown earlier in the section are then proved in Coq.

Example 1. Consider the transition system graphically depicted in Figure 2, which computes in the variable s the sum of natural numbers up to m. We use it as a running example. To encode it in Coq we define a type `Location`

[5] Coq code is shown in `teletype` font mixed with some usual mathematical symbols ($\forall$, $\rightarrow$, etc) for better readability. Coq notions are introduced gradually via examples.

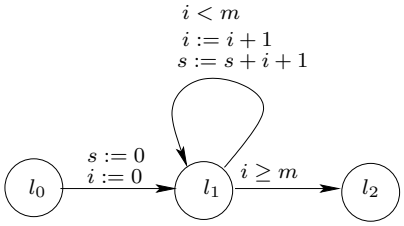

Fig. 2. Running example: sum up to m.

with the constants `l0`, `l1` and `l2` and define the type `State` to be the Cartesian product `Location*nat*nat*nat`. This is interpreted as concrete states being quadruples consisting of a location and three natural numbers. The concrete transition relation is then inductively defined to encode the three "arrows" in Figure 2 (say, `start`, `loop` and `stop`), and their effect on the state variables:

```
Inductive trans: State → State → Prop :=
|start: ∀ m s i, trans(l0,m,s,i) (l1,m,0,0)
|loop: ∀ m s i, i < m → trans(l1,m,s,i) (l1,m,s+i+1,i+1)
|stop: ∀ m s i, i ≥ m → trans(l1,m,s,i) (l2,m,s,i).
```

3 A Proof System for Reachability-Logic Formulas

3.1 Reachability Logic

In this section we generalize RL - currently, a formalism for specifying programming language semantics and a language-parametric program logic - to transition systems, and provide it with a sound and relatively-complete proof system. Abstract Symbolic Execution notions from Section 2 (concrete and abstract states, concrete and abstract transitions, and the relations between them) are assumed.

Definition 1. *A* path *is a finite sequence* $\tau \triangleq s_0 \to \cdots \to s_{n-1} \to s_n$. *We denote by Paths the set of paths. The* length $len(\tau)$ *of a path* τ *is the number of transitions occurring in* τ. *For* $0 \leq i \leq len(\tau)$ *we denote by* $\tau(i)$ *the* i*-th state in the path* τ. *A path is* complete *if the last state on* τ *has no successor w.r.t.* $\to$. *We denote by comPaths the set of complete paths. For* $q \in S^{\#}$ *we let* $Paths(q) \triangleq \{\tau \in Paths \mid \tau(0) \in \gamma(q)\}$, *and* $comPaths(q) \triangleq \{\tau \in comPaths \mid \tau(0) \in \gamma(q)\}$. *If* $len(\tau) \geq k$ *then we denote by* $\tau|_{k..}$ *the suffix of the path* τ *starting at state* $\tau(k)$.

Example 2. In the transition system graphically depicted in Figure 2, the path $(l_0, 1, 0, 0) \to (l_1, 1, 0, 0) \to (l_1, 1, 1, 1) \to (l_2, 1, 1, 1)$ is also a complete path.

Definition 2. *An RL formula is a pair* $l \backsim r$ *with* $l, r \in S^{\#}$. *We let* $lhs(l \backsim r) \triangleq l$ *and* $rhs(l \backsim r) \triangleq r$. *A path* τ satisfies *a formula* $l \backsim r$, *denoted by* $\tau \models l \backsim r$, *if* $\tau \in comPaths(l)$ *and there exists* $k \in \{0 \dots len(\tau)\}$ *such that* $\tau(k) \in \gamma(r)$. *An RL formula is* valid, *denoted by* $\models l \backsim r$ *if for all* $\tau \in comPaths(l)$, $\tau \models l \backsim r$.

$$\textsf{[Impl]}\ \frac{G \vdash G'}{G \vdash \{\langle b, i, l \lhd r\rangle\} \cup G'}\ \textit{if}\ l \sqsubseteq r$$

$$\textsf{[Split]}\ \frac{G \vdash \{\langle b, i+1, l_1 \lhd r\rangle, \langle b, i+1, l_2 \lhd r\rangle\} \cup G'}{G \vdash \{\langle b, i, l \lhd r\rangle\} \cup G'}\ \textit{if}\ l \sqsubseteq l_1 \vee l_2$$

$$\textsf{[Step]}\ \frac{G \vdash \{\langle \mathit{true}, i+1, l' \lhd r\rangle\} \cup G'}{G \vdash \{\langle b, i, l \lhd r\rangle\} \cup G'}\ \textit{if}\ l \wedge f \sqsubseteq \bot, \overset{\#}{\rightarrow}(l) \sqsubseteq l'$$

$$\textsf{[Circ]}\ \frac{G \cup \{\langle \mathit{false}, 0, l' \lhd r'\rangle\} \vdash \{\langle \mathit{true}, i+1, l'' \lhd r\rangle\} \cup G'}{G \cup \{\langle \mathit{false}, 0, l' \lhd r'\rangle\} \vdash \{\langle \mathit{true}, i, l \lhd r\rangle\} \cup G'}\ \textit{if}\ l \sqsubseteq l', r' \sqsubseteq l''$$

Fig. 3. A 4-rule proof system.

Thus, RL formula validity means a formula is *satisfied by all complete paths* induced by a given transition relation (globally assumed in this section and hereafter). This notion naturally specify the partial correctness of possibly parallel/nondeterministic systems, because, as far as validity is concerned, infinite paths are not relevant, but, on the other hand, all complete paths are relevant.

Example 3. For the transition system in Figure 2, consider the RL formula $(l = l_0 \wedge m \geq 0) \lhd (l = l_2 \wedge s = m \times (m+1)/2)$. It specifies that, on all complete paths starting in l_0 with $m \geq 0$, the sum of natural numbers up to m is computed in the variable s. Note that complete paths are those ending in l_2. Thus, the above formula expresses the functional correctness of this simple system.

3.2 Proof system

We now present a proof system for proving the validity of sets of RL formulas.

Final states The validity of RL formulas depends on *final* states (without successors by transition). We assume an abstract state f such that s is a final state iff $s \in \gamma(f)$. For example, in the transition system in Figure 2, $f \triangleq (l = l_2)$.

Rules. For technical reasons (related to the system's soundness proof) we shall consider *indexed* formulas $\langle b, i, l \lhd r\rangle$,i.e., triples consisting of a Boolean, a natural number and an RL formula. There are four rules in our proof system (cf. Fig. 3), all of which define deduction between sets of indexed formulas. We describe the effect of the proof system on formulas, ignoring the indexes for now, as they are only there to support the soundness proof (which we explain afterwards).

The first rule says that if the left hand-side l of a formula is in the $\sqsubseteq$ relation with its right-hand side r then the formula is eliminated from the current set of goals (i.e., it is considered proved: this makes sense since such formulas are trivially valid). The second rule allows one to decompose a formula whose left-hand-side is a disjunction, into two formulas, each of which takes one of the disjuncts as its left hand-side. Note that over-approximations are allowed. The third rule essentially replaces a formula's left-hand side by its image by the abstract transition function; again, over-approximations are allowed. The final rule is what makes the system able to deal with unbounded behaviour in the specification under proof. It says that, if the left hand-side l of a formula $l \lhd r$ in the current set of goals is in the $\sqsubseteq$ relation with the left-hand-side l' of a

formula $l' \lhd r'$ from the initial set of goals, then $l \lhd r$ can be replaced by $l'' \lhd r$, for any over-approximation l'' of r'. That is, the formula $l' \lhd r'$ served locally as an "acceleration" of the abstract transition function in the proof of $l \lhd r$.

A proof in our proof system is a finite sequence of rule applications that eventually eliminates all formulas (in a given set of formulas to be proved).

Definition 3 (proof). *Assume a set G of indexed RL formulas of the form $\langle false, 0, l \lhd r\rangle$. A* proof *is a sequence $G_0 \ldots, G_n$ such that for all $i \in 0 \ldots n-1$, $G \vdash G_{i+1}$ is obtained from $G \vdash G_i$ by applying (bottom-up) one of the rules of the proof system; and $G_n = \emptyset$. We let hereafter in this section $\mathcal{G} \triangleq \bigcup_{0 \leq i \leq n} G_i$.*

Example 4. We prove the following set of formulas on the transition in Fig. 2, ignoring the indexes to simplify notations. This proof will also serve as an illustration of the strategy for the relative completeness of our proof system.

$G = \{(l = l_0 \wedge m \geq 0) \lhd (l = l_2 \wedge s = m \times (m+1)/2),$
$(l = l_1 \wedge 0 \leq i \leq m \wedge s = i \times (i+1)/2) \lhd (l = l_2 \wedge s = m \times (m+1)/2)\}$

The first of the two formulas is the transition system's functional correctness. The second formula was chosen such that its left-hand side $l = l_1 \wedge 0 \leq i \leq m \wedge s = i \times (i+1)/2$ is an "invariant" at location l_1 starting from $l = l_0 \wedge m \geq 0$. Like in Hoare logic, proving partial correctness in RL uses invariants; unlike Hore logic, however, we are not here bound by the syntax and semantics of a particular programming language but reason at the abstract level of transition systems.

Let $G_0 \triangleq G$. We apply [Step] to each the above formulas and get

$G_2 = \{(l = l_1 \wedge i = 0 \wedge s = 0 \wedge m \geq 0) \lhd ((l = l_2 \wedge s = m \times (m+1)/2),$
$((l = l_1 \wedge 0 \leq i \leq m \wedge s = i \times (i+1)/2) \vee (l = l_2 \wedge i = m \wedge s = i * (i+1)/2))$
$\lhd (l = l_2 \wedge s = m \times (m+1)/2)\}$

The first of the above formulas was obtained by "moving" its left-hand side[6] from l_1 to l_2, assigning i and s to 0 in the process. The second formula's left-hand side is actually a disjunction, because from l_1 one can "stay" in l_1 (hence the first disjunct) or "move" to l_2 (giving rise to the second disjunct).

We thus apply the [Split] rule in order to split the second formula in two:

$G_3 = \{(l = l_1 \wedge i = 0 \wedge s = 0 \wedge m \geq 0) \lhd (l = l_2 \wedge s = m \times (m+1)/2),$
$(l = l_1 \wedge 0 \leq i \leq m \wedge s = i \times (i+1)/2) \lhd (l = l_2 \wedge s = m \times (m+1)/2),$
$(l = l_2 \wedge i = m \wedge s = i * (i+1)/2)) \lhd (l = l_2 \wedge s = m \times (m+1)/2)\}$

The last of the above formulas can be eliminated by [Impl] since its left hand-side implies (or is $\sqsubseteq$-smaller than) its right-hand side. This leaves in G_4 the first two formulas of G_3. Next, we note that the second formula in G_4 (also second in G_3) can be eliminated by [Circ], using itself (actually, its instance in G_0); and the first formula in G_4 (also first in G_3) can likewise be eliminated by [Circ] using the same formula in G_0 as above - since $(l = l_1 \wedge i = 0 \wedge s = 0 \wedge m \geq 0) \sqsubseteq (l = l_1 \wedge 0 \leq i \leq m \wedge s = i \times (i+1)/2)$ holds. Each of the two eliminations above leave a copy of the formula $(l = l_2 \wedge s = m \times (m+1)/2) \lhd (s = m \times (m+1)/2)\rangle$, which we finally eliminate by [Impl], leaving us with an empty set of formulas to prove.

Thus, we have a proof of the initial set of formulas according to Definition 3.

[6] Note that rules of our proof system never change a formula's right-hand side.

Soundness We now deal with the soundness of our proof system: a set of indexed formulas that has a proof is a set of valid indexed formulas. Hereafter we assume a set of indexed formulas G that has a proof according to Definition 3. In informal explanations we often simply say "formula" instead of "indexed formula".

The soundness proof is nontrivial because it cannot be performed by induction on the length of derivations in the proof system (which would be the more-or-less standard way to proceed). The reason is the [Circ] rule, which is not sound by itself: this rule can prove *any* formula, valid or not, when used in an unrestricted way (just apply [Circ] to the formula using itself and then [Impl]). The [Circ] rule is crucial: without it the proof system is essentially abstract symbolic execution and can only deal with bounded-length paths, which is not enough[7]. This same phenomenon happens in all other versions of proof systems for RL we know of.

However, a proof by induction *on the length of paths* can be performed for the subsystem consisting of the rules [Impl] and [Step]. The idea is that formulas eliminated from the conclusion are satisfied by a given path, whenever formulas added to hypotheses (by the [Step] rule) hold on shorter paths.

Unfortunately, this does not work when including the [Circ] and [Split] rules: induction on shorter paths does not work, and neither does induction on proof length. This is where indexes of formulas come into play: their role is to allow us to combine the path-length well-founded order in a *lexicographic product* with well-founded orders of the index components. Pairs $(\tau, \langle b, j, l \lhd r\rangle) \in \mathit{comPaths} \times \mathcal{G}$[8] *such that* $\tau \in \mathit{comPaths}(l)$, are ordered iff: either the path lengths are ordered by $<$; or, in case the path lengths are equal, the Boolean components of indexes are ordered by $\mathit{false} < \mathit{true}$; or, when both path lengths and Boolean components are equal, the natural-number component of indexes (upper bounded by the proof's length) are ordered by $>$. This is a lexicographical product of three well-founded relations; it is thus itself well-founded, and it enables an induction that proves the soundness of our proof system. We now state our main lemma that we prove according to this well-founded induction and whose corollary is soundness.

Lemma 1. *For all* $\langle b, j, l \lhd r\rangle \in \mathcal{G}$ *and* $\tau \in \mathit{comPaths}(l)$*, it holds that* $\tau \models l \lhd r$.

We say a set G of indexed formulas is valid if for every $\langle i, l \lhd r\rangle \in G$, $\models l \lhd r$. As a direct consequence of Lemma 1 and of the definition of validity we obtain:

Theorem 1 (Soundness). *If G has a proof then G is valid.*

3.3 A Completeness Result

Soundness is important but is still only about half of the story, because, e.g., a system that proves nothing is sound. We still need to demonstrate the ability of our system to actually "prove something". This has two aspects: a theoretical one, called *completeness*, which says that all valid formulas can, in principle, be proved (possibly up to "oracles" dealing with certain sub-tasks); and a practical one: applying the proof system on concrete examples. We note that in our case theory

[7] We note that, even though the validity of RL formulas only involves finite paths, over the (infinite) set of all such paths the *sup* of path lengths is typically infinite.

[8] Here, the set of formulas $\mathcal{G}$ is that introduced at the end of Definition 3.

and practice are not disjoint, as the completeness result suggests a practical strategy for actually proving all valid formulas. We deal with the theoretical aspect in this section, and illustrate the practical one in the next section.

The completeness result relies on the abstract states being precise enough, as formalised by the following assumption. Given a finite set of indexed RL formulas G to be proved, and a set of terminal states encoded by an abstract state f, the assumption says that certain co-reachable sets of states depending on f and on right-hand sides of formulas in G are exactly represented by abstract states:

Assumption 3 (Coreachable States as Abstract States) *For each $\langle b, i, l \lhd r\rangle \in G$, there exists $coReach_f^+(r) \in S^\#$ with $\gamma(coReach_f^+(r)) = \{s \in S \mid \forall \tau \in Paths(s).\ \tau(len(\tau)) \in \gamma(f)$ implies $(\exists k) 1 \leq k \leq len(\tau)$ such that $\tau(k) \in \gamma(r)\}$.*

That is, the concretisation of the abstract state $coReach_f^+(r)$ consists of the set of concrete states from which *all paths τ of length ≥ 1 that "end up" in a terminal state in $\gamma(f)$ are bound to "encounter" a state in $\gamma(r)$ along the way.*

This assumption may appear strong, but it is trivially satisfied by our encoding of *ASE* in Coq; essentially, this is because abstract states are predicates over concrete states and the concretisation function is just predicate satisfaction, and therefore one can define $coReach_f^+(r) \triangleq \lambda s : State. \forall \tau \in Paths(s). f(\tau(len(\tau))) \rightarrow (\exists k) 1 \leq k \wedge k \leq len(\tau) \wedge r(\tau(k))$, which satisfies the above assumption.

The next definition introduces the notion of *terminator* for an RL formula.

Definition 4. *The RL formula $\mathcal{I} \lhd r$ is a* terminator *for the RL formula $l \lhd r$ if $\mathcal{I} \wedge f \sqsubseteq \bot$, $\overset{\#}{\rightarrow}(\mathcal{I}) \sqsubseteq \mathcal{I} \vee r$ and $l \sqsubseteq \mathcal{I}$.*

Note that this essentially makes $\mathcal{I}$ an inductive invariant starting from states in the concretisation of l: the inclusion $\overset{\#}{\rightarrow}(\mathcal{I}) \sqsubseteq \mathcal{I} \vee r$ says that $\mathcal{I}$ is an invariant starting from concretisations of l except in states that are concretisations of r.

The important thing about terminators is that, together with the formulas they are terminators of, they constitute (as indexed formulas) a set that has a proof in our proof system (according to Definition 3). This is stated by Lemma 3 below, whose proof is kept in the main paper body, since it shows the general strategy for proving an RL formula together with its terminator. We note that the proof shown in Example 4 is just an instance of this general strategy.

We shall first need a monotony property of the abstract transition function:

Lemma 2. *For all $q, q' \in S^\#$ with $q \sqsubseteq q'$, $\overset{\#}{\rightarrow}(q) \sqsubseteq \overset{\#}{\rightarrow}(q')$ holds.*

Lemma 3. *If $\mathcal{I} \lhd r$ is a terminator for $l \lhd r$, then $\{\langle false, 0, l \lhd r\rangle, \langle false, 0, \mathcal{I} \lhd r\rangle\}$ has a proof.*

Proof. Let $G_0 \triangleq \{\langle false, 0, l \lhd r\rangle, \langle false, 0, \mathcal{I} \lhd r\rangle\}$ From $l \sqsubseteq \mathcal{I}$ (cf. Definition 4 of terminators) we obtain using Lemma 2, that $\overset{\#}{\rightarrow}(l) \sqsubseteq \overset{\#}{\rightarrow}(\mathcal{I})$ and thus $\overset{\#}{\rightarrow}(l) \sqsubseteq \mathcal{I} \vee r$. Using this information we apply [Step] to the first formula and obtain $G_1 = \{\langle true, 1, \mathcal{I} \vee r \lhd r\rangle, \langle false, 0, \mathcal{I} \lhd r\rangle\}$ and then by similarly applying [Step] to the second formula, $G_2 = \{\langle true, 1, \mathcal{I} \vee r \lhd r\rangle\}$. We then apply the rule [Split], which generates the set $G_3 \triangleq \{\langle true, 2, \mathcal{I} \lhd r\rangle, \langle true, 2, r \lhd r\rangle\}$. Using [Impl] this reduces to $G_4 \triangleq \{\langle true, 2, \mathcal{I} \lhd r\rangle\}$. Next, we apply [Circ] to obtain $G_5 \triangleq \{\langle true, 3, r \lhd r\rangle\}$) We can thus apply [Impl] again, which leaves us with $G_6 = \emptyset$: the proof is done.

Remark 1. Lemma 2 essentially reduces the proof of any RL formula $l \lhd r$ to the discovery of predicates $\mathcal{I}$ that over-approximate the left-hand side l and are either stable under the transition relation or "end up" in the right-hand side r of the formula. One can note similarities with induction loop invariants for proving pre/post conditions of loops in Hoare logics. The difference is, again, that here we can reason on abstract transition systems instead of concrete programs.

The next question is how to obtain such predicates. This is where abstract states of the form $\mathit{coReach}_f^+(r)$ whose existence is stated by Assumption 3 occur.

Lemma 4. *If $\models l \lhd r$ and $l \wedge r \sqsubseteq \bot$, $\mathit{coReach}_f^+(r) \lhd r$ is a terminator for $l \lhd r$.*

We thus obtain our completeness result, which essentially says that valid RL formulas can be included in a possible larger set of formulas that has a proof.

Theorem 2 (Completeness). *Assume a finite set G of $(\mathit{false}, 0)$-indexed RL formulas. If $\models G$ then there exists a finite set $G' \supseteq G$ that has a proof.*

Proof. Set $G' \triangleq G \cup \bigcup_{\langle \mathit{false},0,l \lhd r\rangle \in G} \{\langle \mathit{false}, 0, \mathit{coReach}_f^+(r) \lhd r\rangle\}$, and use Lemmas 3 and 4 to build proofs for pairs $\{\langle \mathit{false}, 0, l \lhd r\rangle, \langle \mathit{false}, 0, \mathit{coReach}_f^+(r) \lhd r\rangle\}$. In case $l \wedge r \not\sqsubseteq \bot$ for some of the formulas in G, first use the [Split] rule to decompose the formulas into $l \wedge \overline{r} \lhd r$ (which do satisfy $(l \wedge \overline{r}) \wedge r \sqsubseteq \bot$) and $l \wedge r \lhd r$ (which are eliminated by [Impl]).

We conclude this section with two observations.

The first observation is that for the ASE framework that we implement in Coq (which, as explained above, satisfies Assumption 3), our completeness result can be expressed more traditionally as *relative completeness*, i.e., completeness relative to an oracle that can decide the validity of implications between state predicates in Coq's logic. Indeed, in such cases we can obtain terminators by "choosing" the appropriate formulas $\mathit{coReach}_f^+(r) \lhd r$ and running the proof system with them; the latter requires the ability to know whether certain abstract states are ordered by $\sqsubseteq$, which, on our Coq implementation of ASE, amounts to check whether the implication between state predicates holds for all concrete states. Since Coq is an interactive system the oracle is here, ultimately, the user.

The second observation is that, even though the completeness result is a theoretical one, it does provide us with a strategy for our proof system: for a given set G of formulas to prove, we need to add the corresponding terminators in order to obtain a proof. It is the user's responsibility to come up with the appropriate terminators, whose left-hand side, as noted in Remark 1 above, generalise inductive loop invariants. Conceptually, this is a generalisation to RL and transition-system of program verification in Hoare logics, where users must produce inductive loop invariants in order to help prove pre/post conditions.

4 Implementation and Case Study

In this section we describe the implementation of our proof system and of its soundness proof in the Coq proof assistant, and the application of the resulting

Coq RL prover to the functional correctness of a security hypervisor model. We also show an excerpt of the C+ARM implementation of the hypervisor and give some figures based on experiments with it, which show that hypervision introduce an acceptable amount of execution-time overhead on operating-system calls.

Coq [3] is a proof assistant that implements a higher-order logic called *the calculus of inductive constructions*. Coq proofs are lambda-terms that can be independently checked by a relatively simple typechecking algorithm, either the one implemented in Coq or external ones. In this sense, Coq proofs are independently-checkable *certificates*; we thus implemented our prof system in Coq in order to obtain a Coq-certified prover for RL formulas. Thus, when we apply the implemented proof system to the hypervisor case study we obtain a high degree of trust in what has been proved, which is quite important in our case since both the soundness proof and the hypervisor proof are quite intricate.

4.1 Proof of Soundness in Coq

The implementation of our proof system in Coq and the soundness proof closely follow their description in this paper. The "paper version" of the proof system is built upon ASE; thus, the Coq version relies on an implementation of ASE in Coq, already shown in Section 2. There is an arbitrary State type for concrete states and a transition relation trans: State → State → Prop, which are Coq *parameters* to be instantiated for given applications (e.g., the security hypervisor). The other ASE notions and assumptions become Coq definitions and lemmas.

For readability reasons we will not be showing much Coq code in this subsection. We discuss, however, some choices that we made for the implementation.

The first choice, already mentioned above, is to implement the proof system and to prove its soundness in a parameterised setting. The second choice is the kind of *embedding* of our logic (RL) and of its proof system in Coq's logic and proof system. We chose *deep embedding* at this level - the only possible choice if we want to prove soundness. The alternative *shallow embedding* would consist in "emulating" our proof system rule by those of Coq's calculus of inductive constructions, which may not be possible and would definitely make a soundness proof impossible, since Coq cannot reason about itself. However, for usability reasons we shallowly embed *abstract states* as *predicates of type* State → Prop *in Coq's own logic*. This is essential for being able to *actually use* the proof system, which requires an effective mechanism to reason about abstract states (e.g/, proving that two abstract states are in the abstract order relation). Letting abstract states reuse the predefined Prop type enables us to reuse many predefined proof tactics that belong with this type; reimplementing all of them, as a deep embedding of abstract states in Coq would require, is essentially unthinkable.

These choices have further consequences for the usability of the proof system, induced by what is and is not allowed by the Prop type. The most important issue is that equality of terms of this type is not decidable in Coq's logic, thus, neither is the equality of abstract states (based on Prop). Thus, in order to implement sets of RL formulas (that are just pairs of abstract states) we cannot use various existing implementations of sets in Coq, which all require a decidable

equality on set elements. As a consequence, for operations involving sets of RL formulas such as those employed by our proof system (e.g., adding/removing an element to/from a set) we had to use Coq *relations* rather than functions, since the corresponding functions require a decidable equality test on set elements.

Using relations rather than functions means that, when the proof system is applied to actual examples, an intensively used operation such as removing a formula from a set won't be performed by automatic function computation, but by interactive proof, which thus requires further effort from the user.

Fortunately, Coq has an elaborate *tactic language* that can automate, e.g., proofs for such auxiliary set operations. One writes a tactic per operation once and for all, thereby relieving the user from proving facts about auxiliary operations over and over again instead on focusing on the system under verification.

Overall, we strike a balance between being able to prove the proof system's soundness and being able to apply it. The Coq development for the soundness proof is about 1500 lines long and it required about one person-month of work.

4.2 Hypervisor Case Study: General Description

We now describe our case study: a security hypervisor for machine code. The main idea for the supervisor is to "scan" machine-code instructions before letting them be executed by a processor in kernel (privileged) mode. Running arbitrary instructions in privileged mode is a security risk since an attacker could, e.g., access memory zones it should not, or even completely "freeze" the processor.

Most instructions are *normal*, i.e., the processor can safely execute them in privileged mode. These typically include arithmetical and logical operations on user-reserved registers. Other instructions are *special*: they present a potential security risk when executed in privileged mode, such as for example instructions that access memory-management data structures. Before they are executed the hypervisor uses its knowledge on the current state of the processor to check whether there is an actual security risk. If this is not the case then the hypervisor passes on the instruction to the processor for execution. If there is a risk, the hypervisors take appropriate actions such as blocking further code execution.

The main *functional correctness* properties of the hypervisor are that (i) *all instructions passed on to the processor are safe to be executed* and (ii) *the hypervised code's semantics is not altered*. That is, the hypervisor does all what it is supposed to do and no more than it is supposed to do. A crucial non-functional property of the hypervisor, which guided its current design, is that hypervision should slow down code execution as little as possible. This excludes, for example, machine-code emulation by the supervisor, since such software emulation is several magnitude-orders slower than hardware execution in a processor. Another unrealistic design is to hypervise and execute instructions one by one: indeed, the alternation between hypervision and execution is a major source of execution-time overhead, because it involves costly operations such as saving and restoring a software image of the processor's state. Thus, the hypervisor must deal with as many instructions as possible before letting them be executed by the processor.

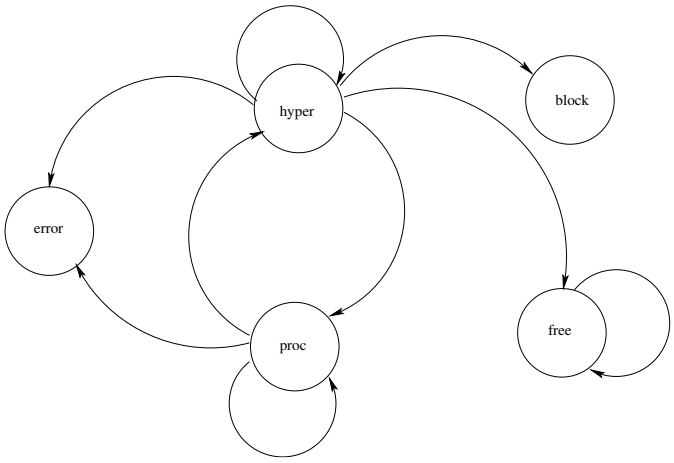

Fig. 4. General structure of hypervisor.

A general graphical depiction of the hypervisor is shown in Figure 4. The locations correspond to several *modes* in which the hypervisor (plus the hypervised system) may be. In *hyper* mode the hypervisor scans instructions. If an instruction is *normal* the hypervisor accepts it and goes on to supervise the next instruction. Now, in many situations, the latter is the actual next instruction in the sequence of instructions constituting a given piece of code. Except, of course, for *jump* instructions, which branch at different tresses than that of the next-in-sequence one. Since the hypervisor does not emulate code execution (this possibility was eliminated for efficiency reasons), it has, in general, no way of knowing after jump instructions which is the next instruction that it should scan.

The idea is then to use the processor execution in a controlled way in order to find out the missing information. The hypervised code is *altered*: the problematic jump instruction is replaced by a so-called *trap* instruction, and the current sequence of hypervised instructions is "flushed" to the processor for execution. In our state machine in in Figure 4 this amounts to switching from the *hyper* to the *proc* mode. When the inserted *trap* execution is reached at execution time, a software image of the processor state is generated, thanks to which the hypervisor "knows" where to continue hypervision after the jump. It can thus go back to *hyper* mode after having restoring the *jump* instruction in order to avoid leaving alterations in the code; then, hypervision of the code is resumed.

The same mechanism is used when a *special* instructions is encountered by the hypervisor: it is replaced by a *trap* instruction, the system goes to *proc* mode for executing the current list of hypervised instructions, and when *trap* is executed, appropriate action is taken: execution is either blocked (which corresponds to switching from *proc* to *blocked* in Figure 4), the system switches back to *hyper* mode, after having restored the *special* instruction in place of the *trap*.

As already noted previously these "switches" from code hypervision to code execution generate much execution overhead. Now, the mode-switches generated by *special* instructions cannot be avoided, otherwise, the hypervisor may violate

its functional-correctness requirements. Switches that can be avoided are only among the ones generated by *jump* instructions the hypervisor could not resolve.

The idea is to save in the hypervisor's state the sequence of instructions already hypervised in the current hypervision phase. In case a jump instruction is encountered, which branches to an instructions in this list, then there is no need to use the above-mentioned trap-execution mechanism. Specifically, if the jump instruction is conditional and only one of the two addresses it may go to is in the already-hypervised instruction list, then hypervision can safely continues at the other address since a second hypervison of an instruction sequence is useless. Moreover, if both tresses a jump instruction may go to are in the current list, then, the need for hypervision is eliminated thereafter, since the code execution will "loop" among instructions already hypervised and known to be "safe". In our state-machine representation this amounts to switching to the *free* mode. The same thing happens for unconditional jumps that go to an instruction in the already-hypervised instruction list. Only in the remaining cases (all jumps go outside the list in question) is that trap-and-execution mechanism required.

This optimisation is not arbitrary, as it deals well standard compilation of while-loops into machine code. The *first time* a loop is encountered it needs, of course, to be hypervised; but when the conditional jump of the loop's compiled code is encountered, which either repeats the loop body of goes after it, the hypervisor need not use a trap-and-execution mechanism to "solve" the condition: all instructions loop body have already been hypervised, thus, hypervision just continues after the loop body. Machine code that we used in benchmarks contains many such loops (active-waiting loops that just decrease a timer counter).

Our hypervisor model has one last mode: *error*, which corresponds to unexpected situations either the hypervisor or the processor detects in a piece of code: for example, binary code that corresponds to no known instruction.

Implementing the hypervisor. Our hypervisor has been implemented for the ARMv6 architecture. Nevertheless, its design allows for a relatively easy evolution for more recent ARM architectures or, with a bit more effort, on different architectures such as Intel's x86/x86_64. The hypervision code has been carefully designed to be as simple as possible and as close as possible to its Coq model.

Listing 1.1. Hypervisor analysis code

```
int found_next = findNextSpecial(state, start, &next, MAX_ITER);
if (found_next) {
  platform.setupTrap(state, next);
  platform.execute(state); // switches to processor mode}

int findNextSpecial(void *state, addr_t start, addr_t *found,
                    const int max_iter) {
  Hypervisor_InstrInfo info;
  addr_t a = 0;
  for (int iter = 0 ; iter < max_iter ; ++iter) {
    platform.instructionInfo(state, a, &info);
    if (is_special_instruction(&info)) {
      *found = a;
      return 1;
    }
    a += info.length;
  }
  return 0;}
```

The hypervisor's code consists in about 4000 lines of C code and 400 lines of ARM assembler. As an example, Listing 1.1 shows C code excerpts from hypervisor primitives. The listing shows the core of the analysis engine. It calls `findNextSpecial` to find the address of the next special instruction and replaces it by a trap that will return to the hypervisor code upon execution. The second function shows how `findNextSpecial` is implemented. For the sake of clarity, debugging messages and some error handling has been stripped from the listings.

The hypervisor has been evaluated on a raspberry pi platform. We performed two kinds of experiments to measure the impact of hypervision on execution time. First, we run the benchmarks in privileged mode. In this mode, the hypervisor needs to deal with all the binary code. Second, we run the same benchmarks as userland tasks on a multi-task guest system. Only the guest system is run in privileged mode and thus it is the only code subject to analysis by our hypervisor.

The benchmarks consist of computation-intensive code: AES encryption, the Dhrystone CPU benchmark, and a serial port peripheral driver code. The AES encryption code is the open source embedded TLS by ARM[9]. The Dhrystone benchmark is the 2.1 version. We measure the overhead using hardware timers. Results given in Table 1 are the overhead ratio of each benchmark, for each execution mode. For example, an overhead value 2 means that the code is two times slower when run under hypervision rather than directly on the hardware.

The results show that the hypervisor introduces a huge overhead when all guest code is run through the hypervisor, with the exception of the driver benchmark that stays reasonable. This can be explained because, as many drivers, its logic relies on busy waiting on the hardware peripheral. Thus, its execution time directly depends on the peripheral rather than on the execution speed of the driver. We do not have results for AES for its execution time was too long to get an actual result. However, a hypervisor is not designed to run arbitrary user code, but OS code with user applications running inside. For such benchmarks the overhead remains under 15%,which makes the approach viable.

Benchmark	Privileged	Userland
AES	–	1.003
Dhrystone	299.7	1.14
Driver	2.4	1.08

Table 1. Execution time overhead introduced by the hypervisor

4.3 Coq model of Hypervisor

The transition-system model of the hypervisor has the general structure of the state machine shown in Figure 4. In addition to the `mode` state-variable, which ranges over the values `hyper`, `proc`, `free`, `block`, and `error`, there are eight other state variables that constitute the State type. Thus, the type `State` is a Cartesian product of nine components. We call these components (by convention[10]):

[9] https://tls.mbed.org

[10] There are no state variables in Coq but, by convention, when we refer to the various components of the `State` type in Coq code we use the above-given names.

- `hi`, a natural number pointing to the current instruction to be analysed;
- `lo`, a natural number pointing to the current instruction to be executed;
- `oldlo`, a natural number memorising the previous version of `lo` (if any);
- `i`, an instruction (more about these below), which memorises the instruction that was replaced by a *trap* instruction when code is instrumented;
- `code`, a list of instructions, modelling the piece of code being hypervised;
- `seen`, a list memorising instructions seen in the current hypervision phase;
- `P`, which models the part of the processor's internal state that is relevant to the control flow of the current `code` being hypervised; since this internal state is a black box, `P` has an arbitrary type called `procState`, with some operations axiomatically defined on (more about this below);
- `len`, a natural number memorising how many instructions were executed.

Before we show in a forthcoming paragraph the Coq encoding of the transition relation we describe some additional artifacts the transition relation refers to.

Instructions and their execution. It is, of course, unrealistic to encode in Coq the whole ARM instruction list. In our model we only need to distinguish between *normal*, *special*, *trap*, and *halt* instructions. In Coq this is encoded as a type `Ins` having one value `norm n` (resp. `spec n` , resp `trap n`) for each natural number `n`, and one value `halt`. The latter instruction represents the end of the current code execution. The effect of the other instruction is modelled by an abstract function (technically, a Coq *parameter*), declared as having the following type:

```
effect : option Ins → procState → procState
```

That is, `effect` takes: a value of type `option Ins` (which is either `Some i`, i.e., an instruction of type `Ins`, or the constant `None`[11]), and a value of type `procState` (i.e., the part of the processor's internal state that is relevant to the control flow of the current code being hypervised); and produces a value of type `procState`.

The actual definition of this function is not given, since we do not model the semantics of ARM instructions. Rather, some of its properties are axiomatised. For example, it is stated that the effect of an instruction replaced (at code analysis/instrumentation time) by a corresponding `trap` instruction is the same as that of the `trap` instruction in question. This is used for proving the requirement that the hypervisor does not alter the semantics of the hypervised code.

Static versus dynamic control flow. By *static* control flow of a piece of code we mean the (typically, incomplete) control-flow information that is known without executing the code. The instructions that jump at constant addresses (or do not jump at all) have such a statically-determined control flow; by contrast, the control flow for instructions that jump at addresses that dynamically depends on `procState` (e.g., on register values) is not determined statically but dynamically.

The static control flow is used during code-analysis phases since, for efficiency reasons, the hypervisor does not emulate code execution in any form; whereas the dynamic control flow is, naturally, used during code-execution phases.

[11] Option types of this kind will also be used by other artifacts of our Coq model.

For static control flow we use a type `Next` consisting of values `none`, `one n`, and `two n m` for natural numbers `n` and `m`, which encodes the three possible cases of an instruction having none, one, or two statically known successors.

The static control flow itself is modelled by an abstractly declared function:

```
nxt : list Ins → nat → Next
```

that, given a list of instructions and a natural-number position, returns the next statically-known address(es) of the next instruction(s) for the instruction at the given position in the given list. Naturally, it is axiomatically specified that, if the given position exceeds the given list length, the value `none` is returned.

The dynamic control flow is also modelled by an abstractly declared function:

```
findNext : list Ins → nat → procState → option nat
```

that, given a list of instructions, a natural-number position, and the processor's state, returns the address of the next dynamically known instruction (if any) for the instruction at the given position in the given list. Naturally, relationships between static and dynamic control have to be stated axiomatically: i.e., if the static control flow is known for a given instruction then its static and dynamic control flows coincide. One last artifact is required by the transition relation.

Code instrumentation. When the hypervisor encounters a special instruction, or a normal instruction for which it cannot statically determine which instruction comes next (e.g., a conditional jump instruction depending on register values), or the halt instruction, it saves the problematic instruction in its state and replaces it with a *trap*. In our Coq model this code instrumentation is modelled by a function `changeIns`. The function itself is quite simple, thus we do not show it here. What's more important is that, *for modelling and verification purposes only*, we adopt the following convention: `halt` is replaced by `trap 0`; and instructions of the form `norm n`, resp. `spec n` are replaced with `trap 2*n+2`, resp. `trap 2*n+1`. This gives us a bijective correspondence between the instructions that are replaced and the trap instructions that replace them, allowing us to axomatically specify that the effects of a replaced instruction and of the instruction replacing it are the same. This property is essential in proving one of the functional correctness requirements: the hypervisor does not alter the code's semantics. What is actually important here is the above-mentioned bijective correspondence and the same-effect property based on it; the way we achieve it (with an encoding based on zero, non-zero even, and odd values) is only a matter of modelling.

Transition relation. The transition relation `trans : State → State → Prop` is defined inductively, just like the one shown in Example 1 but significantly more complex. To illustrate it we show the transitions that originate in `proc` mode.

1. the current instruction is normal, next instruction is statically known and was not seen in current analysis: move to next instruction. Hereafter, the predicate `In` tests the presence of an element in a list, `::` constructs lists, and `nth` returns the nth element of a list (or `None` if the element does not exist).

```
∀ k lo hi oldlo i code pos seen P len, nth code hi = (Some(norm k))→
nxt code hi = (one pos)→ ¬ In pos (hi::seen) →
trans (hyper, lo, hi, oldlo, i, code, seen, P, len)
(hyper, lo, pos, oldlo, i, code, (hi :: seen), P, len)
```

2. current instruction is normal, next instruction is statically known and already seen in current analysis: no more hypervision needed, go to `free` mode.

```
∀ k lo hi oldlo i code pos seen P len, nth code hi = (Some(norm k))→
nxt code hi = (one pos)→ In pos (hi::seen) →
trans (hyper,lo, hi, oldlo ,i,code, seen, P, len)
(free, lo, pos, oldlo, i, code, (hi :: seen), P, len)
```

3. current instruction is a conditional branching, left branch has already been seen, right branch is new : continue hypervision in the right branch.

```
∀ k lo hi oldlo i code l r seen P len, nth code hi = (Some(norm k))→
nxt code hi = (two l r)→ In l (hi::seen) → ¬ In r (hi::seen) →
trans (hyper,lo, hi, oldlo ,i,code, seen, P, len)
(hyper, lo, r, oldlo,i,code, (hi :: seen), P, len)
```

4. a case symmetrical to the previous one, obtained by inverting left and right.
5. current instruction is a conditional branching, both left and right branches have already been seen: no more hypervision needed, go to `free` mode.

```
∀ k lo hi oldlo i code l r seen P len, nth code hi = (Some(norm k))→
nxt code hi = (two l r)→ In l (hi::seen) →In r (hi::seen) →
trans (hyper,lo, hi, oldlo ,i,code, seen, P, len)
(free, lo, hi, oldlo, i, code, (hi :: seen), P, len)
```

6. current instruction is normal, next instruction is not statically known: memorise it, set current instruction to corresponding `trap`, and go to `proc` mode.

```
∀ k lo hi oldlo i code pos l r seen P len,nth code hi=(Some(norm k))→
(nxt code hi = none ∨ (nxt code hi = one pos ∧¬ In pos (hi::seen)) ∨
(nxt code hi = two l r ∧¬ In l (hi::seen) ∧¬ In l (hi::seen))) →
trans (hyper,lo, hi, oldlo ,i,code, seen, P, len)
(proc,lo,hi,oldlo,(norm k),(changeIns code hi (trap(2*k+2))),seen,P,len)
```

7. current instruction is special: memorise it, set current instruction to the corresponding `trap` instruction, and go to `proc` mode.

```
∀ k lo hi oldlo i seen P len,nth code hi = (Some(spec k)) →
trans (hyper,lo, hi, oldlo ,i,code, seen, P, len)
(proc,lo,hi,oldlo,(spec k),(changeIns code hi (trap(2*k+1))),seen,P,len)
```

8. current instruction is halt: memorise it, set current instruction to the corresponding `trap` instruction, and go to `proc` mode.

```
∀ k lo hi oldlo i seen P len,nth code hi = (Some(spec k)) →
trans (hyper,lo, hi, oldlo ,i,code, seen, P, len)
(proc,lo,hi,oldlo,halt,(changeIns code hi (trap 0)),seen,P,len)
```

There are nine other such similarly-defined transitions between the other modes of the transition system; we do not list them all here due to lack of space.

4.4 Coq Proof of Hypervisor's Functional Correctness

The functional correctness of the hypervisor is expressed as two two Coq theorems. The first theorem states an invariance property, saying that, *at all times*, the hypervisor does not let special (i.e., potentially dangereous) instructions be executed by the processor. The second theorem states a partial-correctness property: when the hypervised code's execution ends, its global effect on the processor's state is the same as as that of the same code running without hypervision. The first property must hold at all times since special instructions may occur at any time; by contrast, the second property is only relevant at the end of the execution of the hypervised code; indeed, due to code instrumentation during hypervision phases the property may not even hold at all times.

All special instructions are hypervised. For this invariance property we define the set of initial states of the system: the initial mode is `hyper` (since code must first be hypervised befor being run), the hypervisor's and processor's instruction pointers `hi` resp. `lo` are set to zero (by convention, the address of the first instruction in the code), etc. We also define a general notion of invariants as state predicates holding in all reachable states; the latter are defined inductively.

```
Inductive reachable : State → Prop :=
|zero : ∀ s, init s → reachable s (*init characterises initial states*)
|succ : ∀ s s', reachable s → trans s s' → reachable s'.
Definition invariant(P:State → Prop) := ∀ s, reachable s → P s.
```

Our invariance property is then stated as the following theorem:

```
Theorem hypervisor_hypervises:
invariant(fun s⇒match s with (mode,lo,_,_,_,code,_,_,_)⇒
(mode=proc∨mode=free)→ ∃ ins,nth code lo = Some ins ∧∀k,ins≠spec k end).
```

The `fun` anonymous-function construction here defines a predicate, which holds for the states whose relevant components: `mode`, instruction pointer `lo`, and `code` (extracted from states by the `match` construction[12]) satisfy the constraint that whenever mode is `proc` or `free`, the current instruction is not a special one.

Hypervisor does not alter global code semantics. For this partial-correctness property we also need to characterise final states (i.e., without successor in the transition relation) since partial correctness deals with executions ending in such states. These are states where the mode is either `proc` or `free` and the current instruction being executed is either `halt` or a `trap 0` instruction that replaces it.

We also need to characterise *un*hypervised code execution, since our property is about comparing it with hypervised execution. We thus inductively define a predicate `run` that "applies" the `effect` function (that abstractly defines the effect of instructions) for a sequence of instructions of a given length, starting and ending at a given address an with given initial and final processor states. Specifically, `run code (first,procInit) len (last,procFinal)` holds if, by executing

[12] Underscores match state components that do not occur in a state predicate.

`len` instructions from `code`, starting at address `first` from initial processor state `procInit`, the address `last` and processor state `procFinal` are reached. The partial-correctness property is expressed as a the validity of an RL formula:

```
Theorem hypervisor_does_not_interfere : Valid(init ◅ (final ∧ fun s →
match s with(_,lo,_,_,_,code,_,P,len)⇒run code (0,procInit) len (lo,P)end))
```

That is, starting from initial states (characterised by state predicate `init` - the one also used for the above invariance property), all executions that terminate end up in a state satisfying (of course) `final` and, moreover, by running the code unsupervised from the initial address zero and initial processor state `procInit`, after `len` instructions (whose value is "extracted" from the final state), the final address `lo` and final processor state `P` also coincide with those of the final state.

Proving the partial-correctness property. We apply the strategy formalised in the proof of Lemma 3 and illustrated in Example 4 on a small scale. Given the formula `init ◅ r` to be proved, we find a state predicate `I` such that the implications `I → ¬ final`, `(absTrans I) → (I ∨ r)`, and `init → I` are valid.

Since in our case (and, we expect, in many others) both `r → final` and `init → ¬ r` are valid, by setting `I' ≜ (I ∧ final) ∨ r` the three original implications amount to `init → I'` and `(absTrans I') → I'`; and since `absTrans` was defined to be the strongest-postcondition predicate transformer, the validity of the last two implications amount to stating that `I'` is an *inductive invariant*.

Thus, proving RL formulas amounts is reduced to discovering inductive invariants. This is, of course, a difficult task for nontrivial transition relations and properties involving quantifier alternation such as the ones at hand. There are, however, systematic techniques for doing this, discussed in the next paragraph. For proving the RL formula of interest an inductive invariant consisting of a conjunction of 30 predicates was found. Fortunately we were able to reuse many (specifically, 23) of the predicates required for proving the invariance property.

Proving the invariance property. For invariance properties one can apply the *invariant-strengthening* technique: attempting to prove a predicate is inductive (Coq definitions follow), and, in case of failure, examining which of the transitions failed to preserve the predicate and deriving new predicates that, taken in conjunction with the original one, are potentially inductive. This typically needs to be iterated many times before an actual inductive conjunction is obtained.

Thus, a naïve application of this systematic principle in Coq quickly becomes unmanageable because of the case-explosion problem: each inductiveness proof generates a large number of subgoals, and when new conjuncts are added to a predicate in attempts to make it inductive, both the new *and old* conjuncts need to undergo the new proof attempt (and all the subsequent ones), which requires the user to re-prove over and over again subgoals that we already proved in earlier attempts. As a consequence proof sizes and user efforts become unmanageable.

We thus propose a more incremental approach. We say a predicate `P` is *conditionally inductive* w.r.t. a list `L` of predicates if, by assuming the conjunction `∧L` holds on pre-states, `P` is preserved by transitions from pre to post-states:

```
Definition ind_cond(P: State → Prop)(L: list(State → Prop)):= ∀ s s',
(∧L) s → P s → trans s s' → P s'.
```

A predicate is thus inductive iff it is conditionally inductive w.r.t. an empty list.

The following lemma exploits conditional inductiveness.

```
Lemma ind: ∀P L,inductive(∧L)→ind_cond P L→inductive (∧(P::L))
```

It is used as follows: assume that a previous attempt at proving `P` inductive failed, and the user came up with the list `L` of predicates for which she "believes" that `inductive (∧(P::L))` can be proved. By using the above lemma, this amounts to proving the conditional inductiveness of `P` w.r.t. `L`, which is typically feasible when `L` is adequately chosen, and then (separately) the inductiveness of of the conjunction `∧(L)`, where `P` not involved any more. Thus, unlike the naïve approach, the "old" predicate `P`, which is itself a typically large conjunction, does not need to be dealt with over and over again when it is further strengthened.

The current Coq development for the case study is about 2900 lines long (in addition to the 1500 for the proof system's soundness and strategy). Most of the two man-months effort for the case study (in addition to one man-month for the proof system's soundness and strategy) were dedicated to proving invariants.

5 Conclusion and Future Work

We introduce in this paper an approach for proving partial-correctness properties and invariance properties for transition systems. We thus generalize symbolic execution and Reachability Logic (RL) from their usual setting (programs) to transition systems, and propose a proof system for RL, for which we prove soundness (on paper and in the Coq proof assistant) as well as a completeness result. The completeness result has practical value as it suggests a strategy for the proof system that, provided with adequate user input in the form of inductive invariants, is certain to succeed on valid RL formulas over a given transition system. The Coq implementation of the soundness proof provides us with a Coq-certified interactive prover for RL, which we applied to a nontrivial case study of a security hypervisor we designed in earlier. The reduction of partial correctness to invariance, and an incremental approach we designed for proving invariants, were helpful in enabling us to complete the case study within reasonable time and effort limits. The C+ARM implementation of the hypervisor and some benchmarks are also briefly presented, which indicate that the hypervisor introduces a limited amount of execution-time overhead on operating-system calls.

The main line of future work is exploiting our RL interactive prover in more general ways that that given by reducing each formula's proof to that of one (typically large) inductive invariant. This technique does work, both in theory and in practice, but it does not result in modular proofs, which our proof system allows in principle; for example, separately proving an RL formula characterising a loop, and thereafter simplifying the proof by replacing the loop by the formula. We are also planning to refine our hypervisor model with further optimisations that we implemented to enhance its performances, and to prove the functional correctness of the refined model using the envisaged modular proof technique.

Appendix: Proofs

Lemma 5. *For all $s \in S$, $q \in S^{\#}$ and $\tau \in Paths$, if $s \in \gamma(q)$ and $s = \tau(0)$ then for any $0 \leq k \leq len(\tau)$, $\tau(k) \in \gamma(Reach^{\#}(q))$.*

Proof. Use $Reach^{\#}(q) \sqsupseteq \overset{\#}{\rightarrow}_k(q)$ for all $k \geq 0$ and prove $\tau(k) \in \overset{\#}{\rightarrow}(q)$ by induction on k. Use the property $s \in \gamma(q)$ *implies* $s' \in \overset{\#}{\rightarrow}(q)$ *for all* s' *such that* $s \rightarrow s'$ of abstract interpretation (cf. Fig 1, left) for establishing the induction step.

Lemma 6. *If $\tau \in Paths(q)$ and $len(\tau) \geq 1$ then $\tau|_{1..} \in Paths(\overset{\#}{\rightarrow}(q))$.*

Proof. Let $\tau \triangleq s_0 \rightarrow s_1 \cdots$ with $s_0 \in \gamma(q)$. Then, $\tau|_{1..}$ is the suffix of τ starting at s_1, and all we need to prove is that $s_1 \in \gamma(\overset{\#}{\rightarrow}(q))$, which is a simple consequence of the fact that the abstract transition function simulates concrete transitions.

Lemma 1 *For all $\langle b, j, l \lhd r\rangle \in \mathcal{G}$ and $\tau \in comPaths(l)$, it holds that $\tau \models l \lhd r$.*

Proof. We define an order $\prec$ on the product $comPaths \times \mathcal{G}$ by: $(\tau, \langle b, j, l \lhd r\rangle) \prec (\tau', \langle b', j', l' \lhd r'\rangle)$ iff either $len(\tau) < len(\tau')$ or ($len(\tau) = len(\tau')$ and $b < b'$) or ($len(\tau) = len(\tau')$ and $b = b'$ and $i' > i$). Since: the ordering of complete paths by length; the $<$ relation on Booleans with *false* $<$ *true*; and the $>$ relation the subset of natural numbers up to n (the length of the proof) are well-founded orders, their *lexicographic product* $\prec$ is a well-founded order as well.

We proceed by well-founded induction on $\prec$. We consider an arbitrary pair $(\tau, \langle b, j, l \lhd r\rangle) \in comPaths \times \mathcal{G}$ such that $\tau \in comPaths(l)$.

Let $s = \tau(0)$, thus, $s_0 \in \gamma(l)$. Since $\langle b, j, l \lhd r\rangle \in \mathcal{G}$ then there is i such that was eliminated at step i, that is, $\langle b, j, l \lhd r\rangle \in G_i \setminus G_{i+1}$. We have four cases:

1. $\langle b, j, l \lhd r\rangle$ was eliminated by [Impl]: then, $l \sqsubseteq r$, and $\tau \models l \lhd r$ holds trivially.
2. $\langle b, j, l \lhd r\rangle$ was eliminated by [Split]: then, $l \sqsubseteq l_1 \vee l_2$ and there are formulas $\{\langle b, j+1, l_1 \lhd r\rangle, \langle b, j+1, l_2 \lhd r\rangle\} \subseteq G_{i+1} \subseteq \mathcal{G}$. From $s_0 \in \gamma(l)$, using Assumption 1 we obtain $s_0 \in \gamma(l_1)$ or $s_0 \in \gamma(l_2)$. Assume $s_0 \in \gamma(l_1)$ - the other case is symmetrical. Then, the pair $(\tau, \langle b, j+1, l_1 \lhd r\rangle) \in comPaths \times \mathcal{G}$ satisfies $\tau \in comPaths(l_1)$, and by definition of $\prec$ we have $(\tau, \langle b, j+1, l_1 \lhd r\rangle) \prec (\tau, \langle b, j, l \lhd r\rangle)$. Using the induction hypothesis, $\tau \models l_1 \lhd r$, i.e., $\tau(k) \in \gamma(r)$ for some k, which implies $\tau \models l \lhd r$ and proves this case.
3. $\langle b, j, l \lhd r\rangle$ was eliminated by [Step]: we first show (†): s_0 is not terminal. For, assuming the contrary, $s_0 \in \gamma(l) \cap \gamma(f) = \gamma(l \wedge f)$ (by Asssumption 1) and therefore $l \wedge f \not\sqsubseteq \bot$, in contradiction with $l \wedge f \sqsubseteq \bot$ required for applying [Step]. We thus have $\tau = s_0 \rightarrow s_1 \rightarrow \cdots$ and $\overset{\#}{\rightarrow}(l) \sqsubseteq l'$ with the added formula $\langle true, j+1, l' \lhd r\rangle \in G_{i+1} \subseteq \mathcal{G}$. From $s_0 \rightarrow s_1$ we obtain using Lemma 6 that $s_1 \in \gamma(\overset{\#}{\rightarrow}(l))$. Using Assumption 1 this implies $s_1 \in \gamma(l')$, thus, $\tau|_{1..} \in comPaths(l')$. Thus, $\tau|_{1..} \in comPaths(l')$, and by definition of $\prec$, we have $(\tau|_{1..}, \langle true, j+1, l' \lhd r\rangle) \prec (\tau, \langle b, j, l \lhd r\rangle)$ and we obtain $\tau|_{1..} \models l' \lhd r$; thus $\tau(k) \models r$ for some $k \geq 1$. Hence, $\tau \models l \lhd r$, which proves this case.

4. $\langle b, j, l \lhd r\rangle$ was eliminated by [Circ]: then, $b = \mathit{true}$ and $l \sqsubseteq l'$, for some $\langle \mathit{false}, 0, l' \lhd r'\rangle \in G$, and then $\langle \mathit{true}, j+1, l'' \lhd r\rangle \in G_{i+1} \subseteq \mathcal{G}$, with $r' \sqsubseteq l''$. From $s_0 \in \gamma(l)$ we obtain $s_0 \in \gamma(l')$. Thus, $\tau \in \mathit{comPaths}(l')$, and then $(\tau, \langle \mathit{false}, 0, l' \lhd r'\rangle) \prec (\tau, \langle \mathit{true}, j, l \lhd r\rangle)$ by the definition of $\prec$, which by induction implies $\tau \models l' \lhd r'$. Thus, $\tau(k) \in \gamma(r')$ for some $k \geq 0$.
   If $k = 0$ then from $\tau(k) \in \gamma(r') \subseteq \gamma(l'')$ we get $\tau \in \mathit{comPaths}(l'')$ and then $(\tau, \langle \mathit{true}, j+1, l'' \lhd r\rangle) \prec (\tau, \langle \mathit{true}, j, l \lhd r\rangle)$ by the definition of $\prec$, which implies $\tau \models l'' \lhd r$ i.e. there is $k' \geq 0$ such that $\tau(k') \in \gamma(r)$, i.e., $\tau \models l \lhd r$.
   Otherwise, $k \geq 1$ with $\tau(k) \in \gamma(r')$ and, again, since $r' \sqsubseteq l''$, $\tau(k) \in \gamma(l'')$. Thus, $\tau|_{k..} \in \mathit{comPaths}(l'')$, and $(\tau|_{k..}, \langle \mathit{true}, j+1, l'' \lhd r\rangle) \in \mathit{comPaths} \times \mathcal{G}$ is in the $\prec$ relation with $(\tau, \langle \mathit{true}, j, l \lhd r\rangle)$, thus, $\tau|_{k..} \models l'' \lhd r$, i.e. there is $k' \geq k$ such that $\tau(k') \in \gamma(r)$, which again implies $\tau \models l \lhd r$.

Thus, for all the possible ways in which $\langle b, j, l \lhd r\rangle \in \mathcal{G}$ can be eliminated during the proof, it holds that for any $\tau \in \mathit{comPaths}(l)$, $\tau \models l \lhd r$. The lemma is proved.

Lemma 2 *For all $q, q' \in S^{\#}$ with $q \sqsubseteq q'$, $\overset{\#}{\rightarrow}(q) \sqsubseteq \overset{\#}{\rightarrow}(q')$ holds.*

Proof. Consider an arbitrary state $s' \in \gamma(\overset{\#}{\rightarrow}(q))$. By Assumption 2 there exists a concrete state $s \in \gamma(q)$ with $s \rightarrow s'$. Since $q \sqsubseteq q'$ we also obtain $s \in \gamma(q')$. Thus, using the fact that the abstract transition function simulates concrete transitions we obtain $s' \in \gamma(\overset{\#}{\rightarrow}(q'))$. Thus, $\gamma(\overset{\#}{\rightarrow}(q)) \subseteq \gamma(\overset{\#}{\rightarrow}(q'))$, i.e., $\overset{\#}{\rightarrow}(q) \sqsubseteq \overset{\#}{\rightarrow}(q')$.

Lemma 4 *If $\models l \lhd r$ and $l \wedge r \sqsubseteq \bot$, $\mathit{coReach}_f^+(r) \lhd r$ is a terminator for $l \lhd r$.*

Proof. We have to show the following:

- $\mathit{coReach}_f^+(r) \wedge f \sqsubseteq \bot$, i.e., $\gamma(\mathit{coReach}_f^+(r) \wedge f) = \emptyset$, which by Assumption 1 amounts to $\gamma(\mathit{coReach}_f^+(r)) \cap \gamma(f) = \emptyset$, i.e., $\gamma(\mathit{coReach}_f^+(r))$ contains no terminal states, which is ensured by Assumption 3 (i.e., if $\gamma(\mathit{coReach}_f^+(r))$ contained a terminal state s then any $\tau \in \mathit{comPaths}(s)$ would be of length 0, contradicting the requirement $(\exists k) 1 \leq k \leq \mathit{len}(\tau)$ *such that* $\tau(k) \in \gamma(r)$).
- $\overset{\#}{\rightarrow}(\mathit{coReach}_f^+(r)) \sqsubseteq \mathit{coReach}_f^+(r) \vee r$: using Assumption 1 this amounts to $\gamma(\overset{\#}{\rightarrow}(\mathit{coReach}_f^+(r)) \subseteq \gamma(\mathit{coReach}_f^+(r)) \cup \gamma(r)$. We choose an arbitrary $s \in \gamma(\overset{\#}{\rightarrow}(\mathit{coReach}_f^+(r))$. Using Assumptions 1 and 2, there is $s' \in \gamma(\mathit{coReach}_f^+(r))$ with $s' \rightarrow s$. Consider thus any path τ starting in s and leading to a state in $\gamma(f)$. The path τ' starting in $s' \in \gamma(\mathit{coReach}_f^+(r))$ obtained by prefixing τ with the transition $s' \rightarrow s$ also leads to (the same state in) $\gamma(f)$. This means that τ' encounters a state in $\gamma(r)$, i.e. there is $k' \in \{1 \ldots \mathit{len}(\tau')\}$ such that $\tau'(k') \in \gamma(r)$. If $k' = 1$ then $\tau'(k') = \tau'(1) = s \in \gamma(r)$. Otherwise, $k' > 1$, and τ has the state $\tau(k'-1) = \tau'(k') \in \gamma(r)$ with $k'-1 \in \{1 \ldots \mathit{len}(\tau))\}$. Thus, in this case, the arbitrarily chosen path path τ starting in s and leading to a state in $\gamma(f)$ has a state $\tau(k) \in \gamma(r)$ for $k \triangleq k'-1 \in \{1 \ldots \mathit{len}(\tau))\}$, i.e., $s \in \gamma(\mathit{coReach}_f^+(r))$. Overall, $s \in \gamma(\mathit{coReach}_f^+(r)) \cup \gamma(r)$: this case is proved.
- $l \sqsubseteq \mathit{coReach}_f^+(r)$, i.e., $\gamma(l) \subseteq \gamma(\mathit{coReach}_f^+(r))$. Let $s \in \gamma(l)$. From $\models l \lhd r$ we obtain that any path τ starting in s and leading to $\gamma(f)$ encounters $\gamma(r)$. Since $l \wedge r \sqsubseteq \bot$, the position at which the path encounters $\gamma(r)$ is in $1 \cdots \mathit{len}(\tau)$, i.e., $s \in \mathit{coReach}_f^+(r)$, which proves this case and the lemma.

References

1. C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12 (10): 576580, 1969.
2. Z. Manna and A. Pnueli The temporal logic of reactive and concurrent systems - specification Springer Verlag, 1992.
3. The Coq proof assistant reference manual. `http://coq.inria.fr`.
4. G. Roşu and A. Ştefănescu. Towards a unified theory of operational and axiomatic semantics. In *ICALP 2012*, Springer LNCS 7392, pages 351–363.
5. G. Roşu and A. Ştefănescu. Checking reachability using matching logic. In *OOPSLA 2012*, ACM, pages 555–574.
6. G. Roşu, A. Ştefănescu, Ş. Ciobâcă, and B. Moore. One-path reachability logic. In *LICS 2013*, IEEE, pages 358–367
7. A. Ştefănescu, Ş. Ciobâcă, R. Mereuţă, B. Moore, T. F. Şerbănuţă, and G. Roşu. All-path reachability logic. In *RTA 2014*, Springer LNCS 8560, pages 425–440.
8. A. Arusoaie, D. Nowak, D. Lucanu, and V. Rusu, A Certified Procedure for RLVerification. In 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNACS'17), 2017.
9. A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. Semantics-based program verifiers for all languages. In *OOPSLA'16*, pages 74–91
10. F. Serman and M. Hauspie. Achieving virtualization trustworthiness using software mechanisms The Tenth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2016), 2016.
11. The $\mathbb{K}$ semantic framework. `http://www.kframework.org`.
12. D. Lucanu, V. Rusu, A. Arusoaie, and D. Nowak. Verifying reachability-logic properties on rewriting-logic specifications. In *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer*, Springer LNCS 9200, pages 451–474.
13. Stephen Skeirik, Andrei Stefanescu, and Jose Meseguer. A constructor-based reachability logic for rewrite theories. Technical report, University of Illinois at Urbana-Champaign, 2017. Available at `http://hdl.handle.net/2142/95770`.
14. G. Roşu. Matching logic. In *RTA 2015*, LIPICS volume 36, pages 5–21.
15. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
16. J. Jaffar, V. Murali, J. Navas, and A. Santosa. TRACER: A symbolic execution tool for verification. In *CAV, 2012*, Springer LNCS 7358, pages 758–766.
17. A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé. Using symbolic execution for verifying safety-critical systems. *SIGSOFT Software Engineering Notes*, 26(5):142–151, 2001.
18. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security 2006*, pages 322–335.
19. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE 2005*, ACM, pages 263–272,
20. C. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE 2010*, ACM, pages 179–180.
21. D. Lucanu, V. Rusu, and A. Arusoaie. A generic framework for symbolic execution: A coinductive approach. *J. Symb. Comput.*, 80:125–163, 2017.
22. C. Rocha, J. Meseguer and C. Muñoz. Rewriting modulo SMT and open system analysis. *J. Log. Algebr. Meth. Program*, 86(1):269-297, 2017.

23. J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. Higher-Order and Symbolic Computation 20(1-2):23–160, 2007.
24. K. Bae, S. Escobar and J. Meseguer, Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. *RTA 2013*, LIPICS volume 21, pages 81–96.
25. L. Moreau and J. Duprat. A Construction of Distributed Reference Counting Technical Report RR1999-18, Ecole Normale Supérieure de Lyon, 1999
26. A. E. McCreight. The Mechanized Verification of Garbage Collector Implementations. PhD thesis, Yale University, 2008.
27. The Krakatoa toolset: `http://krakatoa.lri.fr`.
28. The Frama-C toolset: `http://www.frama-c.com`.
29. The Why3 tool: `http://why3.lri.fr`.
30. The CompCert project: `http://compcert.inria.fr`.
31. Adam Chlipala. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. *ACM Sigplan Notices* 48(9):391–402, 2013.
32. The CoqPL workshop series: `http://conf.researchr.org/series/CoqPL`.
33. B. Pierce. Software Foundations. `http://softwarefoundations.cis.upenn.edu`.
34. A. Chlipala. Formal Reasoning About Programs. `http://adam.chlipala.net/frap`.
35. B. Moore and G Rosu. Program Verification by Coinduction. Technical report `http://hdl.handle.net/2142/73177`, University of Illinois at Urbana Champaign, 2015.
36. The DeepSpec project: `http://deepspec.org`.
37. P. Barham, B. Dragovic K. Fraser, S. Hand, T. Harris A. Ho, R. Neugebauer, I. Pratt and A. Warfield Xen and the art of virtualization. In *SOSP 2003*, ACM, pages 164–177.
38. E. Bugniond, S. Devine, K. Govil and M. Rosenblum Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transaction on Computer Systems (TOCS)* 15(4):412–447,1997.
39. The Qemu toolset: `http://www.qemu.org`.
40. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977*, ACM, pages 238–252.