Proving Partial-Correctness and Invariance Properties of Transition-System Models

Vlad Rusu, Gilles Grimaud, Michaël Hauspie

Proving Partial-Correctness and Invariance Properties of Transition-System Models

Vlad Rusu
Inria, Lille, France
Email: Vlad.Rusu@inria.fr

Gilles Grimaud
University of Lille, France
Email: Gilles.Grimaud@univ-lille.fr

Michael Hauspie
University of Lille, France
Email: Michael.Hauspie@univ-lille.fr

***Abstract*—We propose a deductive verification approach for proving partial-correctness and invariance properties on transition-system models. Regarding partial correctness, we generalise the recently introduced formalism of Reachability Logic, currently used as a language-parametric logic for programs, to transition systems. We propose a sound and relatively complete proof system for the resulting reachability logic. The soundness of the proof system is formally established in the Coq proof assistant, and the mechanised proof provides us with a Coq-certified Reachability-Logic prover for transition-system models. 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 invariance verification; for the latter we propose an incremental technique in order to deal with the case-explosion problem that affects it. All these techniques were instrumental in enabling us to prove, within reasonable time and effort limits, that the nontrivial algorithm implemented in security hypervisor that we designed in earlier work meets its expected functional requirements.**

I. Introduction

Partial correctness and invariance are among the most important functional-correctness 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 several program-verification tools.

In this paper we generalise the verification of partial-correctness and invariance properties, from programs, to transition-system models. This enables the formal verification of the given class of properties in earlier software-design stages (i.e., algorithms) rather than in later ones (i.e., programs).

Our motivation is the well-known fact that the longer it takes to uncover flaws in software, the more it costs to fix them.

How can one specify such properties on transition systems, and how can one verify them? One possibility to consider is that of Hoare logics [1], specifically designed for proving partial correctness and invariance. However, 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.

One could also express the properties of interest in temporal logic [2] and use a model checker for temporal-logic formulas on transition systems. However, model checkers are limited to (essentially) finite-state transition systems (up to state abstractions), a limitation we want to avoid. Yet another option would be to use the Temporal Logic of Actions (TLA) [3]: in order to prove properties of infinite-state transition systems one uses Isabelle/TLA$^+$ [4], an axiomatisation of the TLA's proof system in the Isabelle proof assistant [5]. A theoretical issue with this approach is that of soundness (*does the proof system only prove semantically valid formulas?*), which is not answered positively in their axiomatic setting. A more practical issue is that, to our best knowledge, only invariants are implemented in Isabelle/TLA$^+$: temporal properties, among which the partial-correctness properties of interest to us, are not yet considered.

Contribution: We shall here use the Coq proof assistant [6], whose expressive logic allows one to encode guest logics and their proof systems, and to formally prove the soundness of the proof systems in question. We express partial-correctness properties by generalising *Reachability Logic* (hereafter, RL) to transition-system models. RL [7] is originally a language-parametric program logic generalising Hoare logics. We propose a new proof system for RL in a transition-system setting and mechanise its soundness proof in Coq, thereby obtaining a Coq-certified prover for validity of RL formulas on transition systems; i.e., Coq ensures that an RL formula deemed valid on a transition system by our prover is truly so.

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 RL formulas that are valid on a given transition system. As far as we know this is the first formulation of completeness as a practically usable proof strategy for valid RL formulas. The strategy reduces partial-correctness verification to invariance verification; for the latter we improve on a standard invariant-strengthening technique that amounts to strengthen a state predicate until it becomes *inductive*, i.e., stable under the transition relation. The improvement is an incremental technique that mitigates the *case explosion* problem known to affect invariant strenghtening: an ever-increasing number of proof goals that need to be proved.

All these techniques (proof system, strategy reducing partial correctness to invariance, incremental invariant strengthening) were instrumental in enabling us to verify a nontrivial example with a reasonable amount of time and effort. The example is a transition-system model of a security hypervisor we designed [8]. 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. This algorithm 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 algorithm fulfills its expected functional requirements: it hypervises all "dangerous" instructions in any given piece of code while not semantically altering the code.

Related Work: In addition to the already-mentioned related work we here provide more related-work information regarding RL, formal program-verification in Coq, and hypervisors.

RL is a formalism designed for expressing the operational semantics of programming languages and for specifying programs in the languages in question. There are several versions of the logic, among which [7] that we here generalise to transition systems. Languages whose operational semantics is specified in RL include those defined in the $\mathbb{K}$ framework [9], 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 certain oracles.

Besides the obvious difference (we work with transition systems, whereas they work with programs *in languages whose operational semantics is also defined in RL*) the most significant difference between our RL proof system and theirs is that ours is mechanised in (and therefore, certified by) Coq, whereas theirs are automatic, but are not certified[1]. Thus, verification in our case is less automatic, but it is more trustworthy.

Another significant difference is that our completeness result is not only a theoretical one but also 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 exists (invariant-strengthening: enriching a given predicate with new conjuncts obtained by analysing why it failed to be inductive).

Formal verification in Coq is a vast field; we here focus on program verification. Major programming languages such as Java and C are the object of formalisations using Coq. The Krakatoa [11] toolset for Java, together with its counterpart Frama-C [12] for C, are front-ends to the Why tool [13], which generates proof obligations to be discharged in Coq (among other back-end provers). Another Coq framework, dedicated to a low-level extensible programming language, is Bedrock [14].

A hypervisor is for an operating system what an operating system is for a process: it performs a *virtualisation* of the underlying hardware. Two kinds of virtualisations 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 "diverting" them to calls to hypervisor primitives. Thus, para-virtualisation modifies the source code of its guests: it can be viewed as a collaboration between guest and hypervisor. The Xen [15] hypervisor is an example of this category. By contrast, full virtualisation does not require a guest's source code to be modified; instead, guest operating systems trigger exceptions when attempting to run privileged instructions, which are then handled by the hypervisor. VMWare [16], Qemu [17], and our comparatively simple hypervisor [8] are examples of this category.

Hypervisors implement nontrivial algorithms, and formally verifying them is an active research field ([18], [20], [19], to name but a few - an exhaustive list of references can be found in [21]). We only verify our hypervisor's algorithm, not its implementation; the counterpart is that our verification effort is comparatively much smaller. Beyond hypervisors, full operating-system kernels have been verified [22], [23], [24].

The Coq development for the hypervisor example is about 2900 lines long (in addition to 1500 lines 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) was spent proving invariants. Coq sources are available at `http://project.inria.fr/rlase`.

II. Preliminary Notions

A *transition system* is a pair $(S, \to)$ where S is a set of *States* and $\to \subseteq S \times S$ is the *transition relation*. One usually writes $s \to s'$ instead of $(s, s') \in \to$. A *path* is a finite sequence of states connected by the transition relation. We denote by *Paths* the set of paths. The *length* $len(\tau)$ of a path τ is the number of transition steps occurring in τ. That is, when $\tau \triangleq s_0 \to \cdots \to s_{n-1} \to s_n$ then $len(\tau) = n$. For $0 \leq i \leq len(\tau)$ we denote by $\tau(i)$ the i-th state in the path τ and by $\tau|_{i..}$ the suffix of the path τ starting at position i. A path is *complete* if the last state on τ is *final*, i.e., there is no s' such that $s \to s'$. We denote by *comPaths* the set of complete paths,

We assume a set $S^{\#}$ of *State predicates*, closed under conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). The fact that a predicate p is satisfied by a state s is denoted $p\,s$. The predicates $\top$ (resp. $\bot$) are satisfied by all (resp. by none of) the states. For state predicates p, q, we write $p \Rightarrow q$ to denote the fact that for all states s, if $p\,s$ then $q\,s$. Finally, the *symbolic transition function* $\to^{\#}$: $S^{\#} \to S^{\#}$ lifts the transition relation to state predicates, and is defined such that for all s, $(\to^{\#}(p))\,s$ iff there exists s' such that $p\,s'$ and $s' \to s$. By slightly abusing the lambda anonymous-function notation, the state predicate $\to^{\#}(p)$ can be defined as $\to^{\#}(p) \triangleq \lambda s.\exists s'.(p\,s') \wedge s' \to s$.

These notions naturally translate to Coq as follows[2]. States have an arbitrary Coq type `State`. The transition relation is defined as a predicate on pairs of states; in Coq this is written as `trans: State → State → Prop`, where `Prop` is Coq's predefined type for logical statements. Paths have the type `list State` obtained by instantiating the Coq generic type `list` with the type `State`, and the additional properties (consecutive states are in the transition relation) are written as

[1]We note that Coq soundness proofs have earlier been achieved for various RL proof systems [7], [10]. Those proofs did not grow into practically usable Coq-certified program-provers, however, because the resulting Coq frameworks are too hard to instantiate, even on the simplest programming languages.

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

$$\begin{tikzcd}
l_0 \arrow[r, "s := 0 \\ i := 0"] & l_1 \arrow[loop above, "i < m \\ i := i+1 \\ s := s+i+1"] \arrow[r, "i \geq m"] & l_2
\end{tikzcd}$$

Fig. 1. Running example: sum up to m.

separate definitions. The Coq predefined functions on lists `nth` and `lastn` return the state at a given position on a path, and the suffix of a path starting at a given position, respectively.

State predicates have the type `SymState : Type := State → Prop`. Then, the conjunction, disjunction, negation, bottom, and top operations and constants on state predicates are defined using, respectively, Coq's predefined symbols ∧, ∨, ¬, `False`, and `True`. Implication $p \Rightarrow q$ translates to Coq as `imp(p q: SymState):Prop:= ∀s, p s→q s`. Finally, the symbolic transition function $\rightarrow^{\#}$ is written using Coq's anonymous-function construction `fun`: `symTran(q:SymState):=fun s⇒∃s',q s'∧tran s' s`.

We use as example a transition system that computes the sum of natural numbers up to m (Fig. 1). To encode it in Coq we define a type `Location` with the constants `l0`, `l1` and `l2` and define the type `State` to be the Cartesian product `Location*nat*nat*nat`. The transition relation is written (`L`, `M` and `R` correspond to the left, middle, and right arrows):

```
Inductive tran: State → State → Prop :=
|L: ∀m s i,tran(l0,m,s,i)(l1,m,0,0)
|M: ∀m s i,i<m→tran(l1,m,s,i)(l1,m,s+i+1,i+1)
|R: ∀m s i,i≥ m→tran(l1,m,s,i) (l2,m,s,i).
```

III. Reachability Logic on Transition Systems

We define in this section the syntax and semantics of RL on transition-system models. First, given a transition system $(S, \rightarrow)$ and $q \in S^{\#}$ we let $Paths(q) \triangleq \{\tau \in Paths \mid q(\tau(0))\}$, and $comPaths(q) \triangleq \{\tau \in comPaths \mid q(\tau(0))\}$.

Definition 1: An RL formula is a pair $l \lhd r$ with $l, r \in S^{\#}$. We let $lhs(l \lhd r) \triangleq l$, $rhs(l \lhd r) \triangleq r$. A path τ satisfies *a formula $l \lhd r$, denoted by $\tau \models l \lhd r$, if $\tau \in comPaths(l)$ and $r(\tau(k))$ for some $0 \leq k \leq len(\tau)$. An RL formula is* valid*, denoted by $\models l \lhd r$ if for all $\tau \in comPaths(l)$, $\tau \models l \lhd r$.*

Thus, satisfaction by a complete path means the initial state of the path satisfies the left-hand side of the formula and some state of the path satisfies its right-hand side. Validity means a formula is satisfied by all complete paths induced by a given transition system (globally assumed hereafter). Validity specifies partial correctness because it disregards infinite paths.

For the transition system in Figure 1, consider the RL formula $(l = l_0) \lhd (l = l_2 \wedge s = m \times (m+1)/2)$. This formula specifies that, on all complete paths starting in l_0, the sum of natural numbers up to the natural-number m is computed in the variable s; i.e., it specifies functional correctness for this simple system. Note that complete paths are those ending in l_2.

$$\text{[Imp]}\frac{H \vdash G}{H \vdash \{\langle b, i, l \lhd r\rangle\} \cup G}\ \textit{if}\ l \Rightarrow r$$

$$\text{[Spl]}\frac{H \vdash \{\langle b, i+1, l_1 \lhd r\rangle, \langle b, i+1, l_2 \lhd r\rangle\} \cup G}{H \vdash \{\langle b, i, l \lhd r\rangle\} \cup G}\ \textit{if}\ l \Rightarrow (l_1 \vee l_2)$$

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

$$\text{[Crc]}\frac{H \cup \{\langle \textit{false}, 0, l' \lhd r'\rangle\} \vdash \{\langle \textit{true}, i+1, l'' \lhd r\rangle\} \cup G}{H \cup \{\langle \textit{false}, 0, l' \lhd r'\rangle\} \vdash \{\langle \textit{true}, i, l \lhd r\rangle\} \cup G}\ \textit{if}\ l \Rightarrow l', r' \Rightarrow l''$$

Fig. 2. Proof system.

We now present a proof system for the validity in RL. We note that a direct proof of validity by induction on path lengths is not feasible because the paths being complete makes the induction hypothesis essentially useless, i.e., complete paths of length $< n$ are no suffix to a complete path of length n.

Final states: The validity of RL formulas depends on *final* states (without successors by transition). Hereafter we shall denote by f a state predicate such that s is a final state iff $f\ s$. For example, in the transition system in Figure 1, $f \triangleq (l = l_2)$.

Rules: For technical reasons (related to the system's soundness) 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. 2).

Each rule transforms a *sequent* (i.e., an expression of the form $H \vdash G$, with the *hypotheses H* and *goals G* being sets of indexed formulas) into another sequent. We describe the effect of the rules on sequents, ignoring the indexes of formulas for now, as they are only there to support the soundness proof (which we explain afterwards). Hereafter we often write "formula" instead of "indexed formula". Each rule in the proof system is applied bottom-up, in the sense that a rule transforms the sequent below its line into the one above its line.

The first rule says that if the left hand-side l implies the right-hand side r then the formula $l \lhd r$ 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 split a formula whose left-hand-side implies a disjunction, into two formulas, each of which takes one of the disjuncts as its left hand-side. Right-hand sides always remain the same. The third rule essentially replaces a formula's left-hand side by its image by the predicate transition function, up to an over-approximation. The final rule is what makes the system able to deal with unbounded-length behaviour in the transition system under proof. It says that, if the left hand-side l of a formula in the current set of goals implies 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 $l' \lhd r'$ (also being proved) served as a "local acceleration" in the proof of another formula $l \lhd r$.

Definition 2 (proof): Assume a set G of indexed RL formulas of the form $\langle \textit{false}, 0, l \lhd r\rangle$. A proof *is a sequence $G_0 \ldots, G_n$ such that for all $i \in \{0 \ldots n-1\}$, $H \vdash G_{i+1}$ is obtained from $H \vdash G_i$ by applying (bottom-up) one of the rules of the proof system; and $H = G_0 = G$, $G_n = \emptyset$. We let $\mathcal{G} \triangleq \bigcup_{0 \leq i \leq n} G_i$.*

We prove the following two formulas on the transition system in Fig. 1, ignoring the indexes in order to simplify notations:

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

with $r \triangleq (l = l_2 \wedge s = m \times (m+1)/2)$. The first of the two formulas is just functional correctness. The second formula was chosen such that, taken in disjunction with $l = l_0$, its left-hand side $l = l_1 \wedge 0 \leq i \leq m \wedge s = i \times (i+1)/2$ is what we shall call a "terminator" for the first formula; in more standard terminology it is an invariant at location l_1 starting from $l = l_0$. Like in Hoare logics, proving partial correctness in RL uses invariants; unlike Hoare logics, we are not here bound by the syntax and semantics of a particular programming language.

We apply [Stp] to each the above formulas and get

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

The first of the above formulas was obtained by "moving" its left-hand side 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 (hence the second disjunct). We thus apply the [Spl] rule in order to split the second formula in two:

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

The last of the above formulas can be eliminated by [Imp] since its left hand-side implies its right-hand side. This leaves the first two formulas. Next, we note that the second formula can be eliminated by [Crc] using the second formula in the initial set of goals; and the first formula can likewise be eliminated by [Crc] using the same initial formula - since $(l = l_1 \wedge i = 0 \wedge s = 0) \Rightarrow (l = l_1 \wedge 0 \leq i \leq m \wedge s = i \times (i+1)/2)$ holds. Each of the two eliminations above leave a copy of the formula $(l = l_2 \wedge s = m \times (m+1)/2) \lhd r$, which we finally eliminate by [Imp], leaving us with no more formulas to prove.

Soundness: We now deal with the soundness of our proof system: a set of indexed formulas that has a proof (according to Definition 2) is a set of valid indexed formulas.

Soundness is nontrivial because it cannot be proved by induction on the length of derivations in the proof system (which would be the standard way to proceed). The reason is the [Crc] rule, which is unsound when used in an unrestricted way: it can prove *any* formula, valid or not (just apply [Crc] to the formula using itself and then [Imp]). The [Crc] rule is crucial: without it the proof system can only deal with bounded-length paths, which is not enough: even though the validity of RL formulas only involves finite paths, the maximum length of the (infinite) set of all such paths is typically unbounded.

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

Unfortunately, this does not work when including the [Crc] and [Spl] 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.

Using the the set of formulas $\mathcal{G}$ introduced at the end of Definition 2, pairs $(\tau, \langle b, j, l \lhd r\rangle) \in comPaths(l) \times \mathcal{G}$ 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 $false < true$; 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 well-founded, and it enables an induction that proves a crucial first step towards soundness:

Lemma 1: for all $(\tau, \langle b, j, l \lhd r\rangle) \in comPaths(l) \times \mathcal{G}, \tau \models l \lhd r$.

A set G of indexed formulas is valid if for every $\langle b, i, l \lhd r\rangle \in G$, the formula $l \lhd r$ is valid. A corollary of Lemma 1 is:

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

IV. Relative Completeness

Soundness is important but is still only half of the story, because a system that proves nothing is (vacuously) sound. We still need to demonstrate the ability of our system to actually prove something. This has two aspects: a theoretical one, called *relative completeness*, which says that all valid formulas can, in principle, be proved (relative to "oracles" for certain sub-tasks); and a practical one: effectively applying the proof system on examples. In our case theory and practice go "hand-in-hand", as the completeness result suggests a strategy for proving all valid formulas. We deal with the theoretical aspect in this section, and illustrate the practical one in the next section.

Definition 3: A state predicate $\mathcal{I}$ is a terminator *for the RL formula $l \lhd r$ if $\mathcal{I} \wedge f \Rightarrow \bot$, $\xrightarrow{\#}(\mathcal{I}) \Rightarrow (\mathcal{I} \vee r)$ and $l \Rightarrow \mathcal{I}$.*

The important thing about terminators is that one can construct proofs for the formulas they are terminators of.

Lemma 2: If $\mathcal{I}$ is terminator for $l \lhd r$ then $\{\langle false, 0, \mathcal{I} \lhd r\rangle\}$ and $\{\langle false, 0, l \lhd r\rangle, \langle false, 0, \mathcal{I} \lhd r\rangle\}$ both have proofs.

Lemma 2 says that a succesful strategy for proving an RL formula $l \lhd r$ consists in the discovering terminator predicates $\mathcal{I}$ and then applying a certain sequence of rules to the set essentially consisting of $l \lhd r$ and $\mathcal{I} \lhd r$. This general strategy is illustrated in the left column on a toy example, and we sucesfully used this same strategy on our hypervisor example.

A natural question is whether for valid formulas $l \lhd r$ such terminator predicates exist. The next definition and lemma provide us with a positive answer (unde minor conditions).

Definition 4 (Coreachability Predicate): $coReach^{+}_{f}(r) \triangleq \lambda s. \forall \tau \in Paths(s). f(\tau(len(\tau)) \rightarrow (\exists k) 1 \leq k \leq len(\tau). r(\tau(k))$.

Lemma 3: If $\models l \lhd r$ and $l \wedge r \Rightarrow \bot$, then $coReach^{+}_{f}(r)$ is a terminator for $l \lhd r$.

In practice, the user typically needs to come up with simpler terminators than coreachability predicates, but for relative completeness (w.r.t. an oracle proving implications between state predicates) coreachability predicates are good enough.

Theorem 2 (Relative Completeness): If G is valid and finite then there exists a finite superset $G' \supseteq G$ that has a proof.

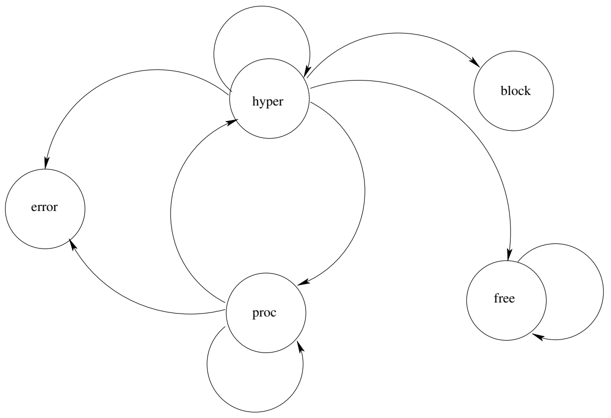

Fig. 3. General structure of hypervisor.

V. Hypervisor Model Example

A. General Description

We now describe our verification example: a security hypervisor for machine code. The main idea is that the supervisor "scans" machine-code instructions before letting them be executed by a processor in kernel mode. Running arbitrary instructions in this mode is clearly a security risk.

Most instructions are *normal*, i.e., the processor can safely execute them in kernel 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 kernel 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. However, if there is a risk, the hypervisor takes appropriate actions such as emulating the instruction, or 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 in kernel mode* and (ii) *the hypervised code's semantics is not altered*. That is, the hypervisor does all what it is supposed to do, but not more. 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 of normal instructions by the supervisor, since the emulation is several magnitude-orders slower than hardware execution in a processor. (However, some special instructions, which are considerably fewer, can/must be emulated in order to avoid security risks.) 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.

A general graphical depiction of the hypervisor is shown in Figure 3. 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, which in most cases is just the actual next-in-sequence instruction in a given piece of code. Except, of course, for *jump* instructions, which typically branch at different adresses than that of the next-in-sequence one. Since the hypervisor only emulates a small subset of the instructions in the code, it has, in general, no way of knowing which is the next instruction that it should scan after a jump instruction.

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 3 this amounts to switching from the *hyper* to the *proc* mode. When the inserted *trap* instruction is reached at execution time, a software image of the processor state is generated, through 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.

The same mechanism is used when a *special* instruction 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 by the hypervisor: execution is either blocked (corresponding to the *blocked* mode in Figure 3), or the special instruction is emulated and the system switches back to *hyper* mode, after having restored the *special* instruction in place of the *trap* to avoid code alteration.

As already stated previously these "switches" from code hypervision to code execution generate much execution overhead. 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 (normal) *jump* instructions.

The idea for this is to save in the hypervisor's state the *list of instructions already hypervised in the current hypervision phase*. In case a jump instruction is encountered, which branches to an instruction 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 continue from the other address since a second hypervison of a "safe" instruction sequence is useless. Moreover, if both adresses a jump instruction may go to are in the current list, then, the need for further hypervision is eliminated altogether, 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 mode switching via trap-and-execution required.

This optimisation is not arbitrary, as it deals well with standard compilation of while-loops into machine code. The *first time* a loop body is encountered it needs, of course, to be hypervised; but when the conditional jump of the loop's compiled code is encountered a second time, the hypervisor need not use a trap-and-execution mechanism to "solve" the condition: all instructions in the loop body have already been hypervised, thus, the processor will be able to safely execute them; for now, hypervision just continues after the loop body.

Our hypervisor model has one last mode: *error*, which corresponds to unexpected situations: for example, binary code that corresponds to no known machine instruction.

B. Coq Model of the Hypervisor

The transition-system model of the hypervisor has the general structure of the state machine shown in Figure 3. 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, in addition to `mode`:

- `hi`: points to the current instruction to be analysed;
- `lo`: points to the current instruction to be executed;
- `oldlo`: the previous version of `lo` (if any);
- `i`: memorises the instruction that was replaced by a *trap*;
- `code`: the piece of code being hypervised;
- `seen`: collects instructions in current hypervision phase;
- `P`: the part of the processor's internal state relevant to the control flow of the current `code` being hypervised;
- `len`: counts how many instructions were executed.

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

a) Instructions and their execution: It is here useless to encode in Coq all machine-code instructions. 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 all other instructions is modelled by an abstractly axiomatised function (technically, a Coq *parameter*) having the signature

```
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`[3]), 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 the function produces a new processor state, of type `procState`.

The actual definition of this function is not written in Coq, since we do not model the semantics of machine-code 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.

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

b) 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 only emulates a small fraction of the executed code; whereas the dynamic control flow is, naturally, used during code-execution phases.

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 a parameter 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, `none` is returned.

The dynamic control flow is also modelled by a parameter:

```
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 axiomatically assumed: i.e., if the static control flow is known for a given instruction then the static and dynamic control flows have to coincide.

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*, which has the effect ob branching execution in the hypervisor, which takes appropriate action (e.g., it emulated a special instruction). In our Coq model code instrumentation is modelled by a function `changeIns`, which 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 the 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 instructions that replace them, allowing us to axomatically specify that, e.g., the global effect of emulating a special instruction is the same as that of executing the instruction in hardware (which is essential for proving that the hypervisor does not alter the code's semantics). What is important here is the bijective correspondence; the way we achieve it is a matter of modelling.

c) Transition relation: The transition relation `trans : State → State → Prop` is defined inductively, just like the one shown in Section II but significantly more complex. To illustrate it we show the following transition, which corresponds to: the currently analysed instruction is normal, the next instruction is statically known and was not seen in current analysis. Then, the hypervisor moves to the next statically-known instruction. Here, 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)
```

There are 17 such similarly-defined transitions; we do not list them all here due to lack of space.

C. Coq Proof of Hypervisor's Functional Correctness

The functional correctness of the hypervisor is expressed as 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: indeed, while a special instruction is being emulated, the property will typically not hold.

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 states reachable from a given state predicate `init`:

```
Inductive reach : State → Prop :=
|init: ∀ s,init s → reach s
|step: ∀ s s',reach s → tran s s' → reach s'.
Definition invariant(P:State → Prop) := ∀ s,
reach 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[4]) satisfy the constraint that whenever instructions are executed (i.e, in modes `proc` or `free`) the currently executed 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 the 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 by code instrumentation.

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 `len` instructions from `code`, starting at address `first` and from initial processor state `procInit`, the address `last` and final processor state `procFinal` are reached.

The partial-correctness property is expressed as:

```
Theorem hypervisor_does_not_alter_semantics :
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 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 2 and illustrated in Section II on a small scale. Given the formula `init ⊲ r` to be proved, we find a state predicate `I` such that `I → ¬ final`, `(symTrans 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'` $\triangleq$ `(I ∧ final) ∨ r` the three original implications amount to `init → I'` and `(symTrans I') → I'`; and since `symTrans` 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.

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

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 success.

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 she 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-states to post-states:

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

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` is not involved any more*. Thus, unlike in 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 in subsequent steps.

VI. Conclusion and Future Work

We introduce in this paper an approach for proving partial-correctness properties and invariance properties for transition systems. We generalise Reachability Logic (RL) from its usual setting (programs) to transition systems, and propose a new proof system for RL in this setting, for which we prove soundness as well as relative completeness. While theoretical in nature, the completeness result also has a practical value as it suggests a strategy for the proof system that is certain to succeed on valid RL formulas over a given transition system. The Coq mechanisation of the soundness proof and of the strategy provides us with a Coq-certified interactive prover for RL. The reduction of partial correctness to invariance, and an incremental approach for proving invariants, were helpful in enabling us to complete a nontrivial illustrative example of a security hypervisor within reasonable time and effort limits.

The main line of future work is exploiting our RL interactive prover in more general ways than the strategy of 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.

References

[1] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12 (10): 576–580, 1969.

[2] Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems - specification Springer Verlag, 1992.

[3] The TLA homepage: https://lamport.azurewebsites.net/tla/tla.html.

[4] K. Chaudhuri, D. Doligez, L. Lamport and S. Merz. A TLA+ proof system. In *Proc. KEAPPA - IWIL Workshops*, 2008, pages 17-37.

[5] The Isabelle proof assistant. https://isabelle.in.tum.de.

[6] The Coq proof assistant reference manual. http://coq.inria.fr.

[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] F. Serman and M. Hauspie. Achieving virtualization trustworthiness using software mechanisms In *10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS'16).*

[9] The $\mathbb{K}$ semantic framework. http://www.kframework.org.

[10] 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.

[11] The Krakatoa toolset: http://krakatoa.lri.fr.

[12] The Frama-C toolset: http://www.frama-c.com.

[13] The Why3 tool: http://why3.lri.fr.

[14] 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.

[15] 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.

[16] 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.

[17] The Qemu toolset: http://www.qemu.org.

[18] D. Leinenbach and T. Santen Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009*, LNCS 5650, pages 806-809.

[19] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome and A. Datta. Design, Implementation and Verification of an Extensible and Modular Hypervisor Framework. In *ISPC 2013*, IEEE, pages 430-444.

[20] A. Blanchard et al. A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C. In *FMICS 2015*, LNCS 9128, pages 15–30

[21] P. Bolignano. Formal Models and Verification of Memory Management in a Hypervisor. PhD, Université de Rennes, 2017.

[22] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch and Simon Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 6(53):107-115, 2010.

[23] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extenisble Architecture for Building Certified Concurrent OS Kernels. *USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 653-669.

[24] M. Dam et al. Formal verification of information flow security for a simple ARM-based separation kernel. In *CCS 2013*, ACM, pages 223-234.

Appendix: Proofs (for reviewing purposes only - not intended for publication)

Lemma 4: 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 we need to prove that $(\overset{\#}{\rightarrow}(q))s_1$, which, by the definition of $\overset{\#}{\rightarrow}$, amounts to proving that there exists s' such that $q\, s'$ and $s' \rightarrow s_1$. Setting $s' \triangleq s_0$ proves the lemma. □

Lemma 1: for all $(\tau, \langle b, j, l \lhd r\rangle) \in comPaths(l) \times \mathcal{G}$, $\tau \models l \lhd r$.

Proof. We define an order $\prec$ on the product $comPaths(l) \times \mathcal{G}$ 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 $(\tau, \langle b, j, l \lhd r\rangle) \in comPaths(l) \times \mathcal{G}$. Let $s = \tau(0)$, thus, $l\, s_0$. 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 [Imp]: then, $l \sqsubseteq r$, and $\tau \models l \lhd r$ holds trivially.
2) $\langle b, j, l \lhd r\rangle$ was eliminated by [Spl]: then, $l \Rightarrow l_1 \vee l_2$ 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 $l\, s_0$we obtain $l_1\, s_0$ or $l_2\, s_0$. Assume $l_1\, s_0$ - 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., $r(\tau(k))$ 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 [Stp]: we first show that s_0 is not terminal. For, assuming the contrary, then $f\, s_0$, and using $l\, s_0$ we obtain $(l \wedge f)s_0$ and therefore $l \wedge f \not\Rightarrow \bot$, in contradiction with the applicaton of [Stp]. We thus have a path $\tau = s_0 \rightarrow s_1 \rightarrow \cdots$ with $len(\tau) \geq 1$. We also have $\overset{\#}{\rightarrow}(l) \Rightarrow l'$ 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 4 that $(\overset{\#}{\rightarrow}(l))\, s_1$, which implies $l'\, s_1$, thus, $\tau|_{1..} \in comPaths(l')$. 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 $(\tau(k))r$ 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 [Crc]: then, $b = true$ and $l \sqsubseteq l'$, for some $\langle false, 0, l' \lhd r'\rangle \in G$, and then $\langle true, j+1, l'' \lhd r\rangle \in G_{i+1} \subseteq \mathcal{G}$, with $r' \sqsubseteq l''$. From $l\, s_0$ we obtain $l'\, s_0$, thus, $\tau \in comPaths(l')$, and $(\tau, \langle false, 0, l' \lhd r'\rangle) \prec (\tau, \langle true, j, l \lhd r\rangle)$ by the definition of $\prec$, which by induction implies $\tau \models l' \lhd r'$. Thus, we have $r'(\tau(k))$ for some $k \geq 0$.
If $k = 0$ then from $r'(\tau(0))$ and $r' \Rightarrow l''$ we get $l''(\tau(0))$, thus, $\tau \in comPaths(l'')$ and $(\tau, \langle true, j+1, l'' \lhd r\rangle) \prec (\tau, \langle true, j, l \lhd r\rangle)$ by the definition of $\prec$, which implies $\tau \models l'' \lhd r$ i.e. there is $k' \geq 0$ such that $r(\tau(k'))$, i.e., $\tau \models l \lhd r$.
Otherwise, $k \geq 1$ with $r'(\tau(k))$ and, again, since $r' \Rightarrow l''$, $l''(\tau(k))$. Thus, $\tau|_{k..} \in comPaths(l'')$, and $(\tau|_{k..}, \langle true, j+1, l'' \lhd r\rangle) \in comPaths(l'') \times \mathcal{G}$ is in the $\prec$ relation with $(\tau, \langle true, j, l \lhd r\rangle)$, thus, $\tau|_{k..} \models l'' \lhd r$, i.e. there is $k' \geq k$ such that $r(\tau(k'))$, 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 $\tau \in comPaths(l)$, $\tau \models l \lhd r$. The lemma is proved. □

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

Proof. Since $G \subseteq \mathcal{G}$ we obtain, using Lemma 1, that for all $(\tau, \langle b, j, l \lhd r\rangle) \in comPaths(l) \times G$, $\tau \models l \lhd r$. But this just means that for all $\langle b, j, l \lhd r\rangle \in G$ and $\tau \in comPaths(l)$, $\tau \models l \lhd r$, which using Def. 1 is just the definition of validity of the set G of indexed RL formulas; the theorem is proved. □

Lemma 2: If $\mathcal{I}$ is a terminator for $l \lhd r$, $\{\langle false, 0, l \lhd r\rangle\}$ and $\{\langle false, 0, l \lhd r\rangle, \langle false, 0, \mathcal{I} \lhd r\rangle\}$ have proofs.

Proof. Let $G_0 \triangleq \{\langle false, 0, \mathcal{I} \lhd r\rangle\}$. We apply [Stp] and obtain $G_1 \triangleq \{\langle true, 1, (\overset{\#}{\rightarrow}\mathcal{I}) \lhd r\rangle\}$. Since $\overset{\#}{\rightarrow}\mathcal{I} \Rightarrow (I \vee r)$ (cf. Def. 3) we apply [Spl] and obtain $G_2 \triangleq \{\langle true, 2, \mathcal{I} \lhd r\rangle, \langle true, 2, r \lhd r\rangle\}$. The second of these formulas is elminated by [Imp] and the first one is eliminated by [Crc] with the single formula in G_0 and then [Imp]. Hence G_0 has a proof.

Wo now consider $G'_0 \triangleq \{\langle false, 0, l \lhd r\rangle, \langle false, 0, \mathcal{I} \lhd r\rangle\}$. From $l \Rightarrow \mathcal{I}$ (cf. Def. 3) we obtain using Lemma 5, that $\overset{\#}{\rightarrow}(l) \Rightarrow \overset{\#}{\rightarrow}(\mathcal{I})$ and thus $\overset{\#}{\rightarrow}(l) \Rightarrow \mathcal{I} \vee r$. By applying [Stp] to the first formula using this overapproximation we get $G'_1 \triangleq \{\langle true, 1, (\mathcal{I} \vee r) \lhd r\rangle, \langle false, 0, \mathcal{I} \lhd r\rangle\}$. Applying [Spl] to the first formula we get $G'_1 \triangleq \{\langle true, 2, \mathcal{I} \lhd r\rangle, \langle true, 2, r \lhd r\rangle, \langle false, 0, \mathcal{I} \lhd r\rangle\}$. The first of the formulas in G'_2 is eliminated by [Crc] with the single formula in G_0 and then [Imp]; the second formula is eliminated by [Imp], and one is left with $\{\langle false, 0, \mathcal{I} \lhd r\rangle\}$, i.e., with G_0, which, as we have seen above, has a proof. Hence G'_0 also has a proof. □

Lemma 5: For all $q, q' \in S^{\#}$ with $q \Rightarrow q'$, it holds that $\overset{\#}{\rightarrow}(q) \Rightarrow \overset{\#}{\rightarrow}(q')$.

Proof. Consider an arbitrary state s' such that $(\overset{\#}{\rightarrow}(q))\, s'$. By definition of $\overset{\#}{\rightarrow}$, there exists a state s with $q\, s$ and $s \rightarrow s'$. Since $q \Rightarrow q'$ we also obtain $q'\, s$. Thus, using again the definition $\overset{\#}{\rightarrow}$ we obtain $(\overset{\#}{\rightarrow}(q'))\, s'$. Thus, $\overset{\#}{\rightarrow}(q) \Rightarrow \overset{\#}{\rightarrow}(q')$: the lemma is proved. □

Lemma 3: If $\models l \lhd r$ and $l \wedge r \Rightarrow \bot$ then $coReach^{+}_{f}(r)$ is a terminator for $l \lhd r$.

Proof. We have to show the following:

- $coReach^{+}_{f}(r) \wedge f \Rightarrow \bot$, i.e., $coReach^{+}_{f}(r)$ is not satisfied by terminal states, which is ensured by Definition 4 (i.e., if $coReach^{+}_{f}(r)$ were satisfied by a terminal state s then any $\tau \in comPaths(s)$ would be of length 0, contradicting $(\exists k) 1 \leq k \leq len(\tau).\, r(\tau(k))$).

- $\overset{\#}{\to}(coReach^+_f(r)) \Rightarrow coReach^+_f(r) \vee r$: We choose an arbitrary state s such that $(\overset{\#}{\to}(coReach^+_f(r))s$. By definition of $\overset{\#}{\to}$, there is s' with $(coReach^+_f(r))s'$ and $s' \to s$. Consider thus any path τ starting in s and leading to a state satisfying f. The path τ' obtained by prefixing τ with the transition $s' \to s$ also leads to the same state satisfying f. This means that τ' encounters a state in satisfying r, i.e. there is $k' \in \{1 \dots len(\tau')\}$ such that $r(\tau'(k'))$. If $k' = 1$ then $\tau'(k') = \tau'(1) = s$ satisfies r. Otherwise, $k' > 1$, and τ has the state $\tau(k'-1) = \tau'(k')$ satisfying r with $k' - 1 \in \{1 \dots len(\tau))\}$. Thus, in this case, the arbitrarily chosen path path τ starting in s and leading to a state satisfying f has a state $\tau(k)$ satisfying r, for $k \triangleq k' - 1 \in \{1 \dots len(\tau))\}$, i.e., $(coReach^+_f(r))s$. Overall, $(coReach^+_f(r)) \vee r)s$: this item is proved.
- $l \Rightarrow coReach^+_f(r)$: Let s be arbitrary chosen such that $l\, s$. From $\models l \lhd r$ we obtain that any path τ starting in s and leading to a state satisfying f encounters a state satisfying r. Since $l \wedge r \Rightarrow \bot$, the position at which the path encounters r is in $\{1 \cdots len(\tau)\}$, i.e., $(coReach^+_f(r))s$, which proves this item and the lemma.

$\square$

Theorem 2 (Relative Completeness): If G is valid and finite then there exists a finite superset $G' \supseteq G$ that has a proof.

Proof.

Set $G' \triangleq G \cup \bigcup_{\langle false, 0, l \lhd r\rangle \in G} \{\langle false, 0, coReach^+_f(r) \lhd r\rangle\}$. We first use the [Spl] rule to decompose each formula in G into $l \wedge \overline{r} \lhd r$ (which do satisfy $(l \wedge \overline{r}) \wedge r \Rightarrow \bot$) and $l \wedge r \lhd r$ (which are eliminated by [Imp]). For each remaining formula, of the form $l \wedge \overline{r} \lhd r$, use the corresponding formula $coReach^+_f(r) \lhd r$ to eliminate it, like in the proof of Lemma 2, using the fact, easily established using Lemma 3, that $coReach^+_f(r)$ is a terminator for $l \wedge \overline{r} \lhd r$. Then, one is left with the set of formulas of the form $coReach^+_f(r) \lhd r$, which are again eliminated like in the proof of Lemma 2. The theorem is proved. $\square$