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 Vlad.Rusu@inria.fr

Gilles Grimaud, Michaël Hauspie

University of Lille {Gilles.Grimaud/Michael.Hauspie}@univ-lille.fr

Abstract

We propose an approach for proving partial-correctness and invariance properties of transition systems, and illustrate it on a model of a security hypervisor.

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 coinductive proof system for the resulting logic, which can be seen as performing an "infinite symbolic execution" of the transition-system model under verification. We embed the proof system in the Coq proof assistant and formally prove its soundness and completeness.

The soundness result provides us with a Coq-certified Reachability-Logic prover for transition-system models. The completeness result, although more theoretical in nature, also has a practical value, as it suggests a proof strategy that is able to deal with all valid formulas on a given transition system.

The complete proof strategy reduces partial correctness to invariance. For the latter we propose an incremental verification technique for dealing with the case-explosion problem that is known to affect it. All these combined techniques were instrumental in enabling us to prove, within reasonable time and effort limits, that the nontrivial algorithm implemented in a simple hypervisor that we designed in earlier work meets its expected functional requirements.

1. Introduction

Partial correctness and invariance are among the most important functional-correctness properties of algorithmic programs. Partial correctness can be 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

[☆]This work was partially funded by the European Celtic-Plus Project ODSI C2014/2-12European Celtic-Plus Project ODSI C2014/2-12.

reachable from a given set of initial states. Such properties have been formalised in several 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 systems, thereby enabling the verification of such properties in earlier software-design stages (algorithms) rather than in later ones (programs). The 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 functional-correctness properties on transition systems, and how can one verify them? One possibility to consider is that of Hoare logics [1], which are specifically designed for dealing with partial correctness and invariance. However, Hoare logics intrinsically require *programs*, as their deduction rules focus on how *instructions* modify state predicates; and in this work 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. Moreover, in order to trust the results of such verifications one has to trust the implementations of model checkers, which are complex pieces of code that may contain soundness bugs.

Another option for proving temporal-logic properties on transition systems, but based on deductive verification, is the TLA+ Proof System [3]. In this system, a proof manager maintains the context of a proof and generates proof obligations that can be discharged using different backend reasoners. Unlike model checking, this approach is not limited to finite-state transition systems, but, like model checking, it has trustwortiness issues: to trust the result of a verification one has to trust a rather complex piece of code (here, the implementations of the proof manager and of the translations to the different backends).

Contribution. We use the Coq proof assistant [5], in which proofs are *certificates* that a third party can independently check by running a simple typechecking algorithm; thus, Coq proofs are as trustworthy as it is nowadays possible.

Coq's 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 [6, 7, 8, 9, 10] is a language-parametric program logic generalising Hoare logics, designed to loosen the syntactical dependencies between Hoare logics and the programming languages they are built upon. We propose a new proof system for RL in a transition-system setting. The proof system is *coinductive*, and can be seen as performing an "infinite symbolic execution" of the transition system under verification. Such executions are, in general, out of reach for automatic verifiers, but they pose no special problems to interactive proof assistants equipped with coinduction.

We embed the proof system in Coq by taking advantage of Coq's coinductive features, and mechanise its soundness proof in Coq, thereby obtaining a Coq-certified prover for validity of RL formulas on transition systems; that is, Coq ensures that an RL formula deemed valid by our prover is truly so.

We also mechanise the completeness of our proof system, which, although more theoretical in nature, also has a practical value: firstly, it suggests a strategy for applying the proof system that is able to deal with all RL formulas that are valid on a given transition system; and secondly, it provides us with a simple and sound mechanism for extending our (minimalistic) proof system with new inference rules, thereby making the proof system easier to use.

The complete proof strategy reduces partial-correctness verification to invariance verification; for the latter we improve on a standard *invariant-strengthening* technique that amounts to strengthening a state predicate until it becomes *stable* under the transition relation. The improvement is an incremental technique that mitigates the *case explosion* problem, which is known to affect invariant strengthening: an ever-increasing number of proof goals that need to be proved.

All these ingredients (sound and complete coinductive 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 simple hypervisor that we designed in earlier work [11]. The hypervisor alternates between a 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 machine code, while not altering the semantics of the code in question.

Comparison with the conference version. The most important difference between the present work and the conference version [12] lies in the RL proof system and the proofs of its soundness and completeness. In [12], we did not use Coq's coinduction (which we were not familiar with, at that time), but implemented a "circular reasoning" in an inductive setting. The soundness proof of the resulting circular proof system was quite complex, requiring a well-founded induction on a lexicographic product of several well-founded relations, including artefacts incorporated in the rules of the proof system just for the purpose of soundness. As a result, the Coq formalisation of the results from [12] is more heavy-weight – 1500 lines, compared to 500 lines for the formalisation of the new results from this paper. Moreover, the soundness and completeness results are cleaner in the new version – the old version proves several formulas at once, which migrate between goals and hypotheses, and the soundness/completeness formulations had to be adapted to accommodate that peculiarity. Overall, thanks to Coq's coinduction we obtain a leaner formalisation of a simpler proof system with cleaner, easier to establish soundness/completeness proofs. Finally, in the conference version, the incremental invariant-strengthening technique and the reduction of RL formulas verification to invariants were only sketched. We provide complete and principled formalisations for those contributions as well.

Related Work. We focus on what we believe is the most relevant related work, regarding RL and its interactions with Coq and with coinduction. We also briefly survey program verification in Coq and system-level formal verification.

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 [9] that we here generalise to transition systems. Languages whose operational semantics is specified in RL include those defined in the $\mathbb{K}$ framework [13], 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. Their verification is then performed using proof systems (of which several versions exist), which are sound, and complete relative to "oracles" performing certain auxiliary tasks.

The coinductive nature of RL is known; in particular, coinduction has been used for defining and proving (on paper) the soundness of RL proof systems [14, 15]. Our main difference with respect to those papers is that they formalise the coinductive aspects of RL based on the classical Knaster-Tarski and Kleene fixpoint theorems. By contrast, in this paper, the coinductive aspects of RL are inherited from those of the host proof assistant - Coq - where coinduction is based on substantially different principles: the Curry-Howard isomorphism, which requires in particular that coinductive proofs are corecursive programs. More generally, all existing RL proof systems that we know of use a "circular reasoning" that allows them, under certain conditions ensuring soundness, to "fold" infinite proofs into finite ones by re-using certain formulas to be proved as hypotheses, which then become available in the proofs of other formulas of even of their own. Our proof system uses the sound circular reasoning provided to us "for free" by Coq's coinduction mechanism; by contrast, all other proof systems we know of (including the one presented in the conference version [12] of this paper) implement their own, specific circular reasoning mechanism. As a result, we obtain possibly the simplest proofs for the meta-theoretical properties of soundness and completeness among all those published to this day.

Another feature that distinguishes us from existing works (with the only notable exception of [16]) is that our approach is completely based on Coq's interactive theorem proving, whereas our colleagues sometimes prove the soundness of their RL proof systems in Coq [9, 17], but afterwards leave the Coq environment and implement their proof systems as automatic program verifiers. Thus, verification in our case is less automatic, but it is more trustworthy[1].

Our work is closest to that of Moore et al. [16]. They develop a coinductive reasoning framework in Coq and apply it to verifying RL formulas on programs in several languages. However, unlike here, in [16] no proof system is used: the semantics of programming languages is used directly. They do show that a

[1] It is perhaps worth repeating that Coq proofs are certificates that can be independently and automatically checked by a third party. Since our approach remains inside the Coq environment, verifications performed with it benefit from such certificates. By contrast, the above-mentioned approaches lose this certificate by exiting the Coq environment after the soundness property is proved. In this sense our approach is more trustworthy than the others.

proof system for RL is a particular instance of their approach, but for theoretical reasons only – in order to give a meaning to the soundness and completeness of their approach. Moreover, we target transition systems, whereas they (as well as all related works presented in this paragraph except [12]) target programs.

Summarising, our approach is as far as we know the only one, among related work regarding RL, which uses a proof assistant and its coinduction mechanisms both for proving the soundness and completeness meta-properties of an RL proof system and for applying the proof system to nontrivial transition-system models.

Formal verification in Coq is a vast field; we here focus on program verification. Major programming languages are targets of Coq formalisations. The Krakatoa [18] toolset for Java, together with its counterpart Frama-C [19] for C, are front-ends to the Why tool [20], which can generate 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 [21]. The already-cited approach [16] is, moreover, language-parametric: it takes as input a Coq formalisation of a programming-language semantics, and produces a coinductive verification framework for the semantics of the language in question.

We use a simple, yet notrivial security hypervisor example to illustrate the feasability of our approach. The focus of the paper is the approach, *not* the example. We nonetheless present some works in the currently very active area of hypervisor verification and more generally of system-level formal verification.

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 [22] 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 [23], Qemu [24], and our comparatively simple hypervisor [11] are examples of this category. Hypervisors implement nontrivial algorithms, and formally verifying them is an active research field ([25, 27, 26], to name but a few – an exhaustive list of references can be found in [28]). Beyond hypervisors, full operating-system kernels have been verified [29, 30, 31].

The Coq development for the hypervisor example is about 2500 lines long, in addition to 500 lines for the formalisation of the theory. Most of the two man-months effort for the example (in addition to one man-month for formalising the theory) was spent proving on discovering and proving invariants. The Coq sources are currently available at `http://project.inria.fr/rlase`.

2. Transition Systems and State Predicates

In this section we introduce some basic ingredients of our approach – transition systems and state predicates – and show their representation in Coq.

A *transition system* is a pair $(S, \mapsto)$ where S is a set of *States* and $\mapsto \subseteq S \times S$ is the *transition relation*. One usually writes $s \mapsto s'$ instead of $(s, s') \in \mapsto$.

A *path* is nonempty, possibly infinite sequence $\tau \triangleq s_0 \mapsto \cdots \mapsto s_n \mapsto \cdots$ of states connected by the transition relation. If the path is finite we require that its last state, say, s, is *final*, i.e., there is no state s' such that $s \mapsto s'$.

We assume as a primitive notion a set $S^\#$ of *State predicates*, which is used as a symbolic representation of possibly infinite sets of states. The set $S^\#$ of state predicates is 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* $\overset{\#}{\to} : S^\# \to S^\#$ lifts the transition relation to state predicates, and is defined as follows: for all s, $(\overset{\#}{\to}(p))\, s$ iff there exists s' such that $p(s')$ and $s' \mapsto s$. By slightly abusing the standard lambda notation for anonymous functions, the state predicate $\overset{\#}{\to}(p)$ is defined as $\overset{\#}{\to}(p) \triangleq \lambda s.\exists s'.p(s') \wedge s' \mapsto s$. Using the same lambda notation we define *final* to be the predicate characterising final states (those without $\mapsto$-successors): $\mathit{final} \triangleq \lambda s.\forall s' \neg(s \mapsto s')$.

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, which are nonempty and possibly infinite sequence of states constrained by the transition relation, require coinductive definitions, which we hereafter give in two steps: we first define nonempty, possibly infinite sequences of states, and then add the constraints induced by the transition relation. Here is the first definition:

```
CoInductive Seq:Type :=
| one:∀s:State, Seq
| add:State → Seq → Seq
```

That is, `Seq` is a type, whose elements (called *inhabitants* in type-theoretic settings) can be either or the form `one s`, for some state `s`, or of the form `add s tau`, for some state `s` and *coinductively defined* `tau` also of the type `Seq`. The fact that the definition is coinductive, as indicated by the `CoInductive` keyword, means that the sequences can be finite or infinite; finite sequences are of the from `add` s_0 `(` $\ldots$`(one` s_n`)...)`, that is, obtained by applying the `add` *constructor* a finite number of times, followed by the `one` constructor. Infinite

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

sequences, on the other hand, are used by applying the `add` constructor an infinite number of times, i.e., of the form `add` s_0 `(` `...(add` s_n `(...)...)`.

We now incorporate the constraints induced by the transition relation to characterise the subset of sequences that represent paths. We first define the head of a sequence, which always exists since sequences are nonempty, by using the *pattern-matching* Coq construction over the above–defined sequences:

```
Definition head(t:Seq):State:= match t with |one s⇒s|add s _⇒s end
```

That is, the `head` of sequences of the form `one s` is `s`, and that of sequence of the form `add s _` is also `s`. Here, `_` denotes an anonymous, irrelevant term. We can now coinductively define a predicate on sequences to characterise paths:

```
CoInductive isPath:Seq→Prop :=
| path_one:∀s, final s → isPath (one s)
| path_add:∀s, trans s (head tau)→ isPath tau→ isPath (add s tau).
```

The above definition reads as follows. The predicate `isPath` holds for two kinds of sequences: the singletons, of the form `one s`, whenever `s` is a final state; and the sequences of the form `add s tau`, whenever there is a `trans`-step between `s` and the head of `tau` and, coinductively, `tau` satisfies `isPath`. Thus, paths are finite or infinite sequences of states connected by the transition relation `trans`, and, for the finite paths, their last state has no `trans`-successors.

We represent state predicates in Coq as inhabitants of the type `SymState := State→Prop`. Then, the conjunction, disjunction, negation, bottom, and top operations and constants on state predicates are defined using, respectively, Coq's predefined symbols $\wedge$, $\vee$, $\neg$, `False`, and `True`. Implication $p \Rightarrow q$ translates to Coq as `imp(p q:SymState):Prop:= ∀s, p s→q s`. The symbolic transition function $\mapsto^{\#}$, and the predicate `final` characterising final states, are written using Coq's anonymous-function construction `fun`: `symTran(q:SymState) := fun s⇒∃s',q s'∧trans s' s`, and `final:=fun s⇒∀s', ¬trans s s'`.

3. Running Example

We now present our running example, which will serve ahead the paper as an illustration of our approach. We specifically take advantage of the example to show a definition of an infinite sequence (whose type is defined in the previous section), to present the limitations of infinite-object definitions in Coq, and to show a first simple coinductive proof, as a preparation to more involved coinductive proofs in subsequent sections.

We consider a simple transition system that computes the sum of natural numbers up to a natural-number m. It is graphically depicted in Fig. 1. Informally speaking, the transition system uses three natural-number variables: i, s, and m, and has three control nodes: l_0, l_1, l_2, with arrows connecting them. Each arrow is possibly labelled by a condition (if absent, the condition is implicitly true) and a set of parallel assignments to the variables. If a variable has no assignment on an arrow its value remains unchanged. After assigning i and s to 0 on the arrow from l_0 to l_1, the consecutive values of the variable i are

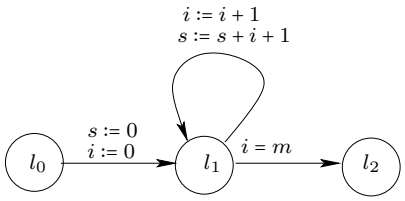

Figure 1: Running example: sum up to m.

accumulated into the variable s by following the self-loop arrow from l_1 to l_1. This accumulation can go on for ever; however, when $i = m$, the transition system also has the possibility to switch to l_2. Hence, we have a nondeterministic transition system, which has both an infinite path (the one self-looping on l_1, on which the variables i and s grow beyond any bound), and finite paths (those that reach l_2, at which point the variable s has accumulated the sum of natural numbers up to m). Simple as it is, this transition system is an infinite-state system, is nondeterministic, and exhibits both infinite and finite paths.

To give it a formal meaning we encode it in Coq. We first define a type `Location` with the constants `l0`, `l1` and `l2` and then define the type `State` to be the Cartesian product `Location*nat*nat*nat`. Finally, the transition relation `trans` is defined as an `Inductive` relation, where the `init`, `loop` and `stop` *constructors* respectively model the left, middle, and right arrows in Fig. 1:

```
Inductive trans:  State → State → Prop :=
|init:  ∀m s i,trans(l0,m,s,i)(l1,m,0,0)
|loop:  ∀m s i,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).
```

Inductive types are similar to coinductive ones, but their inhabitants are finite. Since we now have a transition relation, let us now define some sequences over it – the type `Seq` of sequences was defined in Section 2. For example, the infinite sequence starting with some values of m, s and i and infinitely looping on constructor `loop` is defined as follows, using the `CoFixpoint` keyword:

```
CoFixpoint infLoop(m s i:nat):Seq:=add(l1,m,s,i)(infLoop m (s+i+1)(i+1))
```

Here, `infLoop` can be seen as recursive function that never terminates; its executions are indeed infinite. Such functions are actually called *co-recursive*.

Restrictions for co-recursive functions in Coq. In order to be accepted by Coq, co-recursive functions have to satisfy a *guardedness condition*: every recursive call must be the direct argument of a constructor, and the constructor in question may only be nested within other constructors, or case-analysis (`match`) expressions, or of anonymous function (`fun`) calls [32]. This requirement is sufficient to ensure that co-recursive definitions actually define a unique object [33]. For example, the co-recursive function `infLoop` above does satisfy the guarded-

ness condition: the recursive call is the direct argument of the constructor `add` of the function's type `Seq`, and the constructor is not nested in other expressions.

A first coinductive proof. The guardedness condition must also be satisfied by Coq coinductive proofs, specifically, by *proof objects* that constitute Coq proofs. We illustrate this on a very simple example – a proof script of a few lines, which can be shown and explained in full. In forthcoming, more involved proofs, we will not be showing Coq scripts – they would be too complex to understand, hence we shall revert to "usual" mathematical notations – but we shall refer to the guardedness condition to ensure that conductive proofs are well-formed.

```
Lemma infLoop_isPath:∀ m s i,isPath(infLoop m s i).
Proof.
cofix CIH.
intros m s i.
rewrite infLoop_unfold.
apply path_add.
* apply loop.
* apply CIH.
Qed.
```

The lemma to be proved states that, for all natural numbers `m`, `s`, and `i`, the term `infLoop m s i`, of type `Seq`, satisfies the `isPath` coinductively-defined predicate shown in the previous section. The proof script for this lemma is enclosed between the keywords `Proof` and `Qed`. Let us go through the script:

- the first line invokes the `cofix` tactic, which merely copies the statement to be proved into a *co-induction hypothesis* here called `CIH`. Of course, the co-inductive hypothesis cannot be used right away, otherwise, Coq would be able to prove any statement, which would make it unsound. The hypothesis `CIH` will be used later, in a sound (i.e., *guarded*) context.
- the second line, `intros m s i`, is a formal logical transformation: it replaces the universally quantified variables in the lemma's statement by fresh homonymous constants. Thus, `m`, `s`,`i` are introduced in the hypotheses, and the goal to be proved is unquantified: `isPath(infLoop m s i)`.
- the third line, `rewrite infLoop_unfold`, replaces `infLoop m s i` by its definition; the goal becomes `isPath(add(l1,m,s,i)(infLoop m(s+i+1)(i+1)))`.
- the fourth line, `apply path_add`, uses the `path_add` constructor of `isPath`; it reduces the current goal to the two conditions under which it can hold:
  - there is a transition between `(l1,m,s,i)` and `(l1,m,s+i+1,i+1)`, the `head` of `infLoop m (s+i+1)(i+1)`.This is proved by `apply loop`, where `loop` is the second constructor of the transition relation `trans`;
  - and, moreover, `isPath(infLoop m (s+i+1)(i+1))` holds. It is now safe to apply the coinductive hypothesis `CIH` created at the beginning

$$\textsf{[One]}\ \frac{r(s)}{\tau \rightsquigarrow r}\ \textit{if}\ \tau \triangleq s$$

$$\textsf{[Now]}\ \frac{r(s)}{\tau \rightsquigarrow r}\ \textit{if}\ \tau \triangleq s \mapsto \tau'$$

$$\textsf{[Later]}\ \frac{\tau' \rightsquigarrow r}{\tau \rightsquigarrow r}\ \textit{if}\ \tau \triangleq s \mapsto \tau'$$

Figure 2: Inference rules for leads-to relation $\rightsquigarrow$.

of the proof, since the present goal has been directly obtained by applying the `path_add` constructor of `isPath`, and nothing else; in other words, the current goal is *guarded* by that constructor. Applying `CIH` completes the proof of this goal and of the lemma.

For simple transition-system models such as the one presented in this section it may be possible to state and prove functional properties directly in an ad-hoc manner using coinduction. However, in order to be scalable an approach needs to be systematic. Such an approach is proposed in the following sections.

4. Reachability Logic on Transition Systems

We define in this section the syntax and semantics of RL on transition systems. We use the definitions introduced in Section 2. We assume a fixed transition system $\mathcal{S} \triangleq (S, \mapsto)$, and denote by $\mathit{Paths}(\mathcal{S})$ the set of its paths. Remember that paths are nonempty, finite or infinite sequences of states connected by the transition relation $\mapsto$, and that last states of finite paths are *final*, i.e., without $\mapsto$-successors. For a path τ, $\mathit{head}(\tau)$ denotes the first state in the sequence τ.

Definition 1 (Path Leads To State Predicate). *A path τ* leads to *a state predicate r whenever $(\tau, r) \in \rightsquigarrow$, where $\rightsquigarrow \subseteq \mathit{Paths}(\mathcal{S}) \times S^{\#}$ is the relation coinductively defined by the inference rules in Figure 2. We write $\tau \rightsquigarrow r$ for $(\tau, r) \in \rightsquigarrow$.*

This definition says that $\tau \rightsquigarrow r$ holds whenever s satisfies r and τ is the singleton state s (rule [One]); or τ is of the form $s \mapsto \tau'$, i.e., it is not a singleton, its head is s, and s satisfies r (rule [Now]); or when τ has the form $s \mapsto \tau'$, and (coinductively) $\tau' \rightsquigarrow r$ (rule [Later]). Informally, $\tau \rightsquigarrow r$ whenever τ is finite and some state on τ satisfies r, or when τ is infinite (without other requirements).

The syntax and semantics of RL on transition systems are defined next:

Definition 2 (RL: Syntax and Semantics). An RL formula is a pair $l \Rightarrow\!\Diamond r$ with $l, r \in S^{\#}$. We let $\mathit{lhs}(l \Rightarrow\!\Diamond r) \triangleq l$, $\mathit{rhs}(l \Rightarrow\!\Diamond r) \triangleq r$. A formula f is *valid*, denoted by $\mathcal{S} \models f$, if for all $\tau \in \mathit{Paths}(\mathcal{S})$, if $\mathit{lhs}(f)(\mathit{head}(\tau))$ then $\tau \rightsquigarrow \mathit{rhs}(f)$.

Validity of an RL formula $l \Rightarrow\!\Diamond r$ means that all paths τ that "start" in the formula's left-hand side, i.e., $l(\mathit{head}(\tau))$, must "lead to" the formula's right hand side: $\tau \rightsquigarrow r$. Thus, for all finite paths τ starting in l, some state on

$$\textsf{[Stp]}\ \frac{\mathcal{S} \vdash (\overset{\#}{\rightarrow}(l')) \Rightarrow \diamondsuit r}{\mathcal{S} \vdash l \Rightarrow \diamondsuit r}\ \textit{if}\ l \Rightarrow (l' \vee r), l' \wedge \textit{final} \Rightarrow \bot$$

Figure 3: Coinductive Proof System for RL.

τ satisfying r is reached; for the infinite paths starting in l validity poses no constraints. We have used Linear Temporal Logic (LTL) notations because, semantically, RL formulas are LTL formulas interpreted on finite paths [34].

Example 1. *For the transition system in Figure 1, consider the RL formula* $(l = l_0) \Rightarrow \diamondsuit (l = l_2 \wedge s = m \times (m+1)/2)$. *This formula specifies that, on all finite paths starting in* l_0*, the sum of natural numbers up to the natural-number* m *is computed in the variable* s*. Note that finite paths are those ending in* l_2*. Thus, the RL formula expresses the functional correctness of this transition system.*

We now present a proof system for the validity in RL. Remember from Section 2 that *final* is the state predicate characterising the final states (without $\mapsto$-successor). For example, in the transition system in Figure 1, $\textit{final} \triangleq (l = l_2)$. Remember also the symbolic transition function, denoted $\overset{\#}{\rightarrow} : S^{\#} \rightarrow S^{\#}$.

The proof system has only one rule (cf. Figure 3). It coinductively defines a relation $\vdash$ between transition systems $\mathcal{S}$ and formulas $l \Rightarrow \diamondsuit r$ on $\mathcal{S}$.

The unique rule of the proof system [Stp] says that, in order to prove $\mathcal{S} \vdash l \Rightarrow \diamondsuit r$, one has to come up with a state predicate l' which is an overapproximation of the "difference" between l and r seen as sets of states – which is written in logical terms as $l \Rightarrow l' \vee r$. Moreover, l' should "contain" no final states. Then, proving $\mathcal{S} \vdash l \Rightarrow \diamondsuit r$ reduces to proving $\mathcal{S} \vdash (\overset{\#}{\rightarrow}(l')) \Rightarrow \diamondsuit r$.

Conceptually, a proof of $\mathcal{S} \vdash l \Rightarrow \diamondsuit r$ can be seen as an "infinite symbolic execution" starting from the state predicate l and attempting to reach r. At each step, one only needs to process the "difference" between l and r, since it is only from that difference that r has not yet been reached. We allow actually over-approximations l' of that difference, perform a symbolic step from l', and continue from there on *ad infinitum* since the proof system is coinductive.

We note that coinductiveness is here an essential feature; a finite, inductive interpretation of the proof system in Figure 3 is not able to prove anything since it just keeps postponing a conclusion forever. This is why in other frameworks (including the conference version of this paper) various "circular reasoning" mechanisms that emulate coinduction have been implemented.

Example 2. *We prove the following formula on the transition system in Fig. 1:*

$f \triangleq (l = l_0) \Rightarrow \diamondsuit (l = l_2 \wedge s = m \times (m+1)/2)$.

Let $\mathcal{I} \triangleq (l = l_0) \vee (l = l_1 \wedge s = i \times (i+1)/2)$*, and let* L, R *denote the left and right-hand sides of* f*. It can be proved that the three following implications hold:*

1. $L \Rightarrow (\mathcal{I} \vee R)$

2. $\mathcal{I} \wedge \mathit{final} \Rightarrow \bot$
3. $\overset{\#}{\rightarrow}(\mathcal{I}) \Rightarrow (\mathcal{I} \vee R)$.

Using these implications we obtain a coinductive proof of $f \triangleq L \Rightarrow\!\diamond R$ as follows:

- *we apply [Stp] with l' being $\mathcal{I}$. The first two implications above allow this. We are left with $\mathcal{S} \vdash (\overset{\#}{\rightarrow}\mathcal{I}) \Rightarrow\!\diamond R$ to prove, which we coinductively assume as a hypothesis. (In Coq this corresponds to applying the* `cofix` *tactic),*
- *we apply [Stp] once more, with l' again being $\mathcal{I}$. The last two implications above allow this. We are left with proving the same goal $\mathcal{S} \vdash (\overset{\#}{\rightarrow}\mathcal{I}) \Rightarrow\!\diamond R$,*
- *since we only applied [Stp] – the unique, recursive constructor of our proof system – it is now safe to use the coinductive hypothesis (in Coq terminology, the current goal is* guarded*). This concludes the proof of $\mathcal{S} \vdash L \Rightarrow\!\diamond R$.*

The coinductive proof was illustrated on an example, but that actually gives a strategy for proving any formula $L \Rightarrow\!\diamond R$ for which a state predicate $\mathcal{I}$ satisfying implications (1-3) above can be found. We later use this observation for actually proving formulas and for establishing the completeness of our proof system.

Soundness. Soundness says that any proved formula is valid:

Theorem 1 (Soundness). *For all RL formulas f, if $\mathcal{S} \vdash f$ then $\mathcal{S} \vDash f$.*

Proof. Let $f \triangleq l \Rightarrow\!\diamond r$. By Definition 2, $\mathcal{S} \vDash f$ iff for all $\tau \in \mathit{Paths}(\mathcal{S})$, if $l(\mathit{head}(\tau))$ then $\tau \rightsquigarrow r$. Hence, we reformulate the theorem's statement as

(†) *For all $l, r \in S^{\#}$, for all $\tau \in \mathit{Paths}(\mathcal{S})$, $\mathcal{S} \vdash l \Rightarrow\!\diamond r$ and $l(\mathit{head}(\tau))$ imply $\tau \rightsquigarrow r$.*

We will be using (†) as a coinduction hypothesis when it is safe to do so. Consider any $l, r \in S^{\#}$ and $\tau \in \mathit{Paths}(\mathcal{S})$ such that $\mathcal{S} \vdash l \Rightarrow\!\diamond r$ and $l(\mathit{head}(\tau))$; we prove $\tau \rightsquigarrow r$. From $\mathcal{S} \vdash l \Rightarrow\!\diamond r$ we obtain that there is $l' \in S^{\#}$ such that $l \Rightarrow (l' \vee r)$ and $l' \wedge \mathit{final} \Rightarrow \bot$ (i.e., l' is not satisfied by final states) and $\mathcal{S} \vdash (\overset{\#}{\rightarrow}(l')) \Rightarrow\!\diamond r$. From $l(\mathit{head}(\tau))$ and $l \Rightarrow (l' \vee r)$ we obtain $l'(\mathit{head}(\tau))$ or $r(\mathit{head}(\tau))$.

- If $r(\mathit{head}(\tau))$ then $\tau \rightsquigarrow r$ holds either by [One] or [Now] (cf. Figure 2), depending on whether the sequence τ is, or is not, a singleton state;
- if $l'(\mathit{head}(\tau))$: we first show that τ is not a singleton state s. For, if this were the case, s whould be final, but as we determined above, l' is not satisfied by any final state. Thus, τ has the form $s \mapsto \tau'$. In this case, the rule [Later] (cf. again Figure 2), *which is a constructor of $\rightsquigarrow$*, reduces what we have to prove: $\tau \rightsquigarrow r$, to $\tau' \rightsquigarrow r$. Since up to this point we only applied case analysis and a constructor of $\rightsquigarrow$, it is safe to use the coinductive hypothesis (†), *applied to $\overset{\#}{\rightarrow}(l'), r \in S^{\#}$ and $\tau' \in \mathit{Paths}(\mathcal{S})$*, in order to prove $\tau' \rightsquigarrow r$. We check that the conditions for applying (†) hold:
  - we have established above that $\mathcal{S} \vdash (\overset{\#}{\rightarrow}(l')) \Rightarrow\!\diamond r$ holds;

– hence, we only have to prove that $(\overset{\#}{\rightarrow}(l'))\,(head(\tau'))$ holds. By definition of $\overset{\#}{\rightarrow}$, $(\overset{\#}{\rightarrow}(l'))(head(\tau'))$ holds iff there exists s' such that $l'\,s'$ and $s' \mapsto head(\tau')$. And such an s' does exist: it is s. Indeed, in the current subcase, $l'\,s$, and τ has the form $s \mapsto \tau'$, and $s \mapsto head(\tau')$ follows from the fact that τ and τ' are paths; This settles the last subcase; the proof by coinduction of soundness is complete.

□

Completeness. Soundness is essential but is still only half of the story, because a proof 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 *completeness*, which says that all valid formulas can be proved; and a practical one: effectively applying the proof system on examples. We deal with the completeness aspect in the rest of this section, and illustrate its practical applications in forthcoming sections.

We first formulate a lemma whose proof, which reproduces word for word the one shown in Example 2, gives a generic strategy for proving RL formulas.

Lemma 1 (Strategy). *Let $f \triangleq l \Rightarrow \diamondsuit r$ be any RL formula. If there is $\mathcal{I} \in S^{\#}$ such that $l \Rightarrow (\mathcal{I} \vee r)$, $\mathcal{I} \wedge final \Rightarrow \bot$, and $\overset{\#}{\rightarrow}(\mathcal{I}) \Rightarrow (\mathcal{I} \vee r)$, then $\mathcal{S} \vdash f$.*

Theorem 2 (Completeness). *For all RL formulas f, $\mathcal{S} \vDash f$ implies $\mathcal{S} \vdash f$.*

Proof. Let $f \triangleq l \Rightarrow \diamondsuit r$. We find a state predicate $\mathcal{I}$ that, if $\mathcal{S} \vDash f$, satisfies the three implications from Lemma 1 and implies $\mathcal{S} \vdash f$. Let us define

$$\mathcal{I} \triangleq \lambda s.\, \neg r(s) \wedge \forall \tau \in Paths(\mathcal{S}).\, s = head(\tau) \longrightarrow \tau \rightsquigarrow r$$

That is, $\mathcal{I}$ is satisfied by states that do not satisfy r, but such that all paths starting in s lead to r. We prove that $\mathcal{I}$ is appropriate for establishing $\mathcal{S} \vdash f$:

1. $l \Rightarrow (\mathcal{I} \vee r)$: let s be an arbitrarily chosen state such that $l(s)$; we have to prove that $(\mathcal{I} \vee r)\,s$ holds. If $r(s)$ the proof is done. Thus, assume $\neg r(s)$, and consider any path τ such that $s = head(\tau)$. From $l(s)$ and $s = head(\tau)$ and $\mathcal{S} \vDash l \Rightarrow \diamondsuit r$ it follows, by Definition 2, that $\tau \rightsquigarrow r$, and by the above definition of $\mathcal{I}$ we have $\mathcal{I}(s)$: the first implication is proved.
2. $\mathcal{I} \wedge final \Rightarrow \bot$: let s be an arbitrarily chosen state such that $\mathcal{I}(s)$; we prove that $final(s)$ is impossible. By the above definition of $\mathcal{I}$, $\neg r(s)$. Consider an arbitrary path τ such that $s = head(\tau)$; again, by definition of $\mathcal{I}$, $\tau \rightsquigarrow r$. Now, the only way $\tau \rightsquigarrow r$ can hold when $\neg r(s)$ holds is (cf rule [Later] in Figure 2) when $\tau = s \mapsto \tau'$ for some path τ'. Hence, s is not final, thus no state satisfies $\mathcal{I} \wedge final$, and our second implication is also proved.
3. $\overset{\#}{\rightarrow}(\mathcal{I}) \Rightarrow (\mathcal{I} \vee r)$: let s' be a state such that $(\overset{\#}{\rightarrow}(\mathcal{I}))(s')$; we have to prove $(\mathcal{I} \vee r)(s')$. By the definition of the symbolic transition function $\overset{\#}{\rightarrow}$, there exists s such that $s \mapsto s'$ and $\mathcal{I}(s)$. By the definition of $\mathcal{I}$, $\neg r(s)$ and for each $\tau \in Paths(\mathcal{S})$ such that $s = head(\tau)$, $\tau \rightsquigarrow r$. There are two subcases:

- if $r(s)'$ then $(\mathcal{I} \vee r)\, s'$, and our implication is proved;
- if $\neg r(s')$: consider any path τ' such that $s' = \mathit{head}(\tau')$. Then, the path $\tau \triangleq s \mapsto \tau'$ is such that $s = \mathit{head}(\tau)$, and, per the above, $\tau \rightsquigarrow r$. We also have $\neg r(s)$, and then the only way $\tau \rightsquigarrow r$ may hold is via the rule [Later] in Figure 2. Thus, $\tau' \rightsquigarrow r$. Summarising, in the case $\neg r(s')$, we get that any path τ' such that $s' = \mathit{head}(\tau')$ satisfies $\tau' \rightsquigarrow r$. Hence, $\mathcal{I}(s')$ by the definition of $\mathcal{I}$, and therefore also $(\mathcal{I} \vee r)(s')$, which completes the proof of the last implication and of the theorem.

□

Discussion. Lemma 1 provides us with a general way of proving any RL formula, provided that a state predicate $\mathcal{I}$ can be found, which satisfies three implications. The proof of the completeness theorem above builds on that result and constructs such a predicate for all valid formulas. Looking more closely at the proof one can notice that the (object-level) state predicate $\mathcal{I}$ encodes meta-level information such as the $\rightsquigarrow$ relation. This is a common approach in relative-completeness proofs; for example, existing completeness proofs for RL use a Gödel-style encoding of reachability using prime-factors representation of natural numbers [9]. In other words, completeness is a strong claim, but it relies on strong assumptions. The assumptions would be unrealistic in an automatic verification setting because of general undecidability results, but here we are in an interactive proof assistant where undecidability is not really an issue.

Adding rules to the proof system. The soundness of our proof system together with Lemma 1 above provides us with one way to establish that RL formulas are valid. Of course, proving validity in this manner is not the only way. The general way amounts to using our proof system together with coinduction.

However, using the proof system is somewhat unpractical: its unique rule [Stp] does "everything at once" (choosing an overapproximation l' of the difference between l and r, performing a symbolic step from the result, and coinductively invoking itself), which, in practice, may be hard to achieve. One may also want smaller, easier-to-use proof steps: for example, a rule for plain logical simplifications (or a strengthening) of a proof goal; or a rule for "splitting" disjunctions; or, a transitivity rule, which also splits one formula into two.

Thanks to soundness and completeness we can include any result about validity as a new inference rule. For example, Lemma 2 below gives three such results; they translate to three new, sound inference rules for our proof system (cf Figure 4), which just replace $\vDash$ with $\vdash$ in the statements of the lemma.

We note that the new rules are not direct, "syntactical" consequence of the proof system's unique rule [Stp]. This is most obvious in the case of the new rules [Spl] and [Tra], each of which reduce one proof goal to two goals; the rule [Stp] does not have any way of its own for doing this[3]. Hence, the new rules do introduce new information in the proof system, thanks to the

[3]We have also tried to prove the soundness of the new rules syntactically, i.e., without

$$[\mathsf{Str}]\ \frac{\mathcal{S} \vdash l' \Rightarrow \Diamond r}{\mathcal{S} \vdash l \Rightarrow \Diamond r}\ \textit{if}\ l \Rightarrow l'$$

$$[\mathsf{Spl}]\ \frac{\mathcal{S} \vdash l_1 \Rightarrow \Diamond r \quad \mathcal{S} \vdash l_2 \Rightarrow \Diamond r}{\mathcal{S} \vdash (l_1 \vee l_2) \Rightarrow \Diamond r}$$

$$[\mathsf{Tra}]\ \frac{\mathcal{S} \vdash l \Rightarrow \Diamond m \quad \mathcal{S} \vdash m \Rightarrow \Diamond r}{\mathcal{S} \vdash l \Rightarrow \Diamond r}$$

Figure 4: Auxiliary Rules for Proof System.

equivalence between $\vdash$ and $\models$ that they exploit (even though, in absolute terms, no expressiveness is added, because $\vdash$ is already complete for $\models$, cf. Theorem 2).

We also note that directly including the new rules in a coinductive proof system would result in unsoundness; in such a proof system, e.g., the rule [Str] could prove any formula, by infinitely reducing $\mathcal{S} \vdash l \Rightarrow \Diamond r$ to itself. Thus, the new proof rules should only be applied finitely many times, for the purpose of proof simplifications; only the original rule [Stp] can be applied with coinduction.

Lemma 2 (Some Properties of Validity). *The following three results hold:*

- *for all* l, l', r, $l \Rightarrow l'$ *and* $\mathcal{S} \models l' \Rightarrow \Diamond r$ *implies* $\mathcal{S} \models l \Rightarrow \Diamond r$;
- *for all* l_1, l_2, r, $\mathcal{S} \models l_1 \Rightarrow \Diamond r$ *and* $\mathcal{S} \models l_2 \Rightarrow \Diamond r$ *implies* $\mathcal{S} \models (l_1 \vee l_2) \Rightarrow \Diamond r$;
- *for all* l, m, r, $\mathcal{S} \models l \Rightarrow \Diamond m$ *and* $\mathcal{S} \models m \Rightarrow \Diamond r$ *implies* $\mathcal{S} \models l \Rightarrow \Diamond r$.

Proof. By definition, $\mathcal{S} \models l \Rightarrow \Diamond r$ iff for all $\tau \in \mathit{Paths}(\mathcal{S})$, $l(\mathit{head}(\tau))$ implies $\tau \rightsquigarrow r$.

For the first item: we note that paths τ satisfying $l(\mathit{head}(\tau))$ are also such that $l'(\mathit{head}(\tau))$, and the conclusion $\tau \rightsquigarrow r$ results from $\mathcal{S} \models l' \Rightarrow \Diamond r$.

For the second item: paths τ satisfying $(l_1 \vee l_2)(\mathit{head}(\tau))$ satisfy $l_1(\mathit{head}(\tau))$ or $l_2(\mathit{head}(\tau))$; $\tau \rightsquigarrow r$ results from the hypothesis $\mathcal{S} \models l_i \Rightarrow \Diamond r$ $(i = 1, 2)$.

For the third item, we note that $\tau \rightsquigarrow r$ iff either τ is infinite, or τ is finite and there is a state s on τ such that $r(s)$. Consider then any path τ such that $l(\mathit{head}(\tau))$. If τ is infinite, then our conclusion $\tau \rightsquigarrow r$ follows. If τ is finite, then from $\mathcal{S} \models l \Rightarrow \Diamond m$ we obtain $\tau \rightsquigarrow m$; specifically, since τ is finite, there is a state s on τ such that $m(s)$. Consider then the suffix τ' of τ starting at s. From $\mathcal{S} \models m \Rightarrow \Diamond r$ we obtain $\tau' \rightsquigarrow r$, and since the path τ' is also finite, there is a state s' on τ' such that $r(s')$, which, since τ is finite, implies our conclusion $\tau \rightsquigarrow r$. □

going via the completeness result, but for some reason Coq's coinduction mechanism based on `cofix` rejected all our attempts. In order to get a better understanding we then tried the same exercice in the Isabelle/HOL proof assistant [4], which unlike Coq, generates Knaster-Tarski-style *coinduction principles* from coinductive definitions; and found an explanation. It amounts to the fact that the coinduction principle of our proof system requires a set of formulas closed under the symbolic transition function, but the new rules do not ensure this.

5. Incremental Invariant Strengthening

This section focuses on a systematic approach for proving invariants. We first show how proving RL formulas essentially reduces to proving that certain strong-enough predicates are *initialised* and *stable* under the transition relation of the transition system under verification. Such predicates are, in particular, *invariants*. We then present the systematic technique of *invariant strengthening*, which amounts to strengthening a predicate that is not stable by including into it new predicates that, at least, "postpone" the instability, and, at best, removes it altogether. We also present a refinement of this technique – *incremental* invariant strengthening, which mitigates the *case-explosion* problem that affects (plain) invariant strengthening. All these techniques were used in the security hypervisor case study, whose verification is described in the next section.

We start with a reformulation of Lemma 1, more convenient to use hereafter:

Lemma 3. *Let $f \triangleq l \Rightarrow\!\Diamond r$ be any RL formula. If there is $\mathcal{J} \in S^{\#}$ such that $l \Rightarrow \mathcal{J}$, $\overset{\#}{\rightarrow}(\mathcal{J}) \Rightarrow \mathcal{J}$, and $\mathcal{J} \wedge \mathit{final} \Rightarrow r$, then $\mathcal{S} \vDash l \Rightarrow\!\Diamond r$.*

Proof. We first note that the symbolic transition function $\overset{\#}{\rightarrow} : S^{\#} \rightarrow S^{\#}$ is monotonous : $q \Rightarrow q'$ implies $\overset{\#}{\rightarrow}(q) \Rightarrow \overset{\#}{\rightarrow}(q')$.

We then prove that (†) $\mathcal{J} \Rightarrow ((\mathcal{J} \wedge \neg \mathit{final}) \vee r)$. Indeed, consider any state s such that $\mathcal{J}(s)$. If $\neg \mathit{final}(s)$ then $(\mathcal{J} \wedge \neg \mathit{final})(s)$, otherwise, $\mathit{final}(s)$ and then $(\mathcal{J} \wedge \mathit{final})(s)$, which implies $r(s)$ thanks to the hypothesis $\mathcal{J} \wedge \mathit{final} \Rightarrow r$. Thus, for any state s, $\mathcal{J}(s)$ implies $((\mathcal{J} \wedge \neg \mathit{final}) \vee r)(s)$, and (†) is proved.

Let $\mathcal{I} \triangleq \mathcal{J} \wedge \neg \mathit{final}$. We prove that $\mathcal{I}$ satisfies the implications in Lemma 1.

- $l \Rightarrow (\mathcal{I} \vee r)$: we have $l \Rightarrow \mathcal{J}$ from the hypotheses, and using (†), $\mathcal{J} \Rightarrow ((\mathcal{J} \wedge \neg \mathit{final}) \vee r)$, we obtain, $l \Rightarrow ((\mathcal{J} \wedge \neg \mathit{final}) \vee r)$, which is exactly $\mathcal{I} \vee r$.
- $\mathcal{I} \wedge \mathit{final} \Rightarrow \bot$: results directly from the definition $\mathcal{I} \triangleq \mathcal{J} \wedge \neg \mathit{final}$ above.
- $\overset{\#}{\rightarrow}(\mathcal{I}) \Rightarrow (\mathcal{I} \vee r)$: we have $\overset{\#}{\rightarrow}(\mathcal{I}) = \overset{\#}{\rightarrow}(\mathcal{J} \wedge \neg \mathit{final})$; $\overset{\#}{\rightarrow}(\mathcal{J} \wedge \neg \mathit{final}) \Rightarrow (\overset{\#}{\rightarrow}(\mathcal{J}))$ by monotonicity, $\overset{\#}{\rightarrow}(\mathcal{J}) \Rightarrow \mathcal{J}$ by hypothesis, $\mathcal{J} \Rightarrow ((\mathcal{J} \wedge \neg \mathit{final}) \vee r)$ by (†), and the latter is exactly $\mathcal{I} \vee r$. The implications in Lemma 1 are proved.

We can now apply Lemma 1 and obtain $\mathcal{S} \vdash l \Rightarrow\!\Diamond r$, and soundness (Theorem 1) ensures $\mathcal{S} \vDash l \Rightarrow\!\Diamond r$: the lemma is proved. □

Lemma 3 thus indeed reduces the proof of $\mathcal{S} \vDash l \Rightarrow\!\Diamond r$ to discovering a *strong-enough* predicate $\mathcal{J}$ (i.e., such that $\mathcal{J} \wedge \mathit{final} \Rightarrow r$), which *is initialised in* l (i.e., $l \Rightarrow \mathcal{J}$) and *stable* under the transition relation (i.e., $\overset{\#}{\rightarrow}(\mathcal{J}) \Rightarrow \mathcal{J}$).

Predicates that are initialised and stable are, in particular, *invariants*, which per the above constitute useful auxiliary properties for proving partial correctness; moreover, they often express relevant functional properties by themselves. (This is the case for the hypervisor case study, discussed in the next section.)

Definition 3 (Reachable States and Invariant From State Predicate). *Consider a transition system $\mathcal{S} = (S, \mapsto)$. Given a state predicate l, the set of states* reachable *from l is the smallest set of states $R \subseteq S$ such that $l(s)$ implies $s \in R$ and for all s, s', if $s \in R$ and $s \mapsto s'$ then $s' \in R$. A state predicate r is an* invariant from l *if for all $s \in R$, $r(s)$. When this is the case we write $\mathcal{S} \vDash l \Rightarrow \Box r$.*

We call *invariance formulas* over a transition system $\mathcal{S} = (S, \mapsto)$ the pairs of the form $l \Rightarrow \Box r$ with $l, r \in S^{\#}$. Like for RL formulas we here use LTL notations.

Lemma 4 (Initialised and Stable State Predicate is an Invariant). *If a state predicate $\mathcal{J}$ satisfies $l \Rightarrow \mathcal{J}$ and $\overset{\#}{\rightarrow}(\mathcal{J}) \Rightarrow \mathcal{J}$ then $\mathcal{S} \vDash l \Rightarrow \Box \mathcal{J}$.*

Proof. We prove by induction on the definition of the set R of states reachable from l that for all s, if $s \in R$, $\mathcal{J}(s)$. In the base case, we have $l(s)$ and therefore $\mathcal{J}(s)$ thanks to the implication $l \Rightarrow \mathcal{J}$. For the inductive state, we have to prove that for all s, s', if $\mathcal{J}(s)$ and $s \mapsto s'$ then $\mathcal{J}(s')$. Now, $\mathcal{J}(s)$ and $s \mapsto s'$ implies, using the definition of the transition function $\overset{\#}{\rightarrow} : S^{\#} \rightarrow S^{\#}$, that $\overset{\#}{\rightarrow}(\mathcal{J})(s')$. Since $\overset{\#}{\rightarrow}(\mathcal{J}) \Rightarrow \mathcal{J}$ we get $\mathcal{J}(s')$, which proves the inductive step and the lemma. □

Lemma 4 is the basis of the method called *invariant strengthening* for proving invariants (to our best knowledge, this method is formal-methods folklore). The goal is to prove $\mathcal{S} \vdash l \Rightarrow \Box r$ for some state predicates l, r. One first verifies $l \Rightarrow r$ (easy) and then one proceeds with attempting to prove $\overset{\#}{\rightarrow}(r) \Rightarrow r$. In case of success, by Lemma 4, $\mathcal{S} \vdash l \Rightarrow \Box r$ has been proved and the process stops. Usually, however, this does not work from the first attempt: r is not strong enough. It is thus strengthened by taking the conjunction with a predicate r_1, provided by the user, who *encodes the reason why r failed to be preserved by some transitions, into a state predicate*[4]. After verifying the easy part $l \Rightarrow r_1$ one attempts to prove $\overset{\#}{\rightarrow}(r \wedge r_1) \Rightarrow (r \wedge r_1)$. In case of success, $r \wedge r_1$ is initialised in l and stable; by Lemma 4, $\mathcal{S} \vDash l \Rightarrow \Box(r \wedge r_1)$, and by Definition 3, for every state s reachable from l, $(r \wedge r_1)(s)$ and, in particular, $r(s)$, meaning $\mathcal{S} \vdash l \Rightarrow \Box r$.

If, however, $\overset{\#}{\rightarrow}(r \wedge r_1) \Rightarrow (r \wedge r_1)$ cannot be proved, the user examines again the reason for this stability failure and encodes this reason in a predicate r_2, then attempts to prove that $r \wedge r_1 \wedge r_2$ is initialised in l and is stable.

The process is, of course, not guaranteed to terminate, but it does terminate whenever the user, having gained insight and inspiration during the process, eventually produces an initialised and stable predicate $r \wedge r_1 \wedge \cdots r_n$.

This *plain* invariant-strengthening method has one important drawback: the predicate that one tries to prove stable grows at each invariant-strengthening

[4]Thus, invariant strengthening is not automatic; it is, however, systematic: it includes the user in a feedback loop, providing information about previous stability failures, which the user has to encode into a state predicate; here, insight and inspiration are essential for success.

$$[\mathsf{End}]\ \frac{}{\mathcal{S} \Vdash l \Rightarrow \Box r}\ \textit{if}\ l \Rightarrow r, \overset{\#}{\rightarrow}(r) \Rightarrow r$$

$$[\mathsf{Cnt}]\ \frac{\mathcal{S} \Vdash l \Rightarrow \Box r'}{\mathcal{S} \Vdash l \Rightarrow \Box r}\ \textit{if}\ l \Rightarrow r, (\overset{\#}{\rightarrow}(r \wedge r')) \Rightarrow r$$

Figure 5: Proof System for the $\Rightarrow\Box$ Relation.

step, which makes it harder and harder for the user to analyse in all possible cases why a given step failed and what the relevant next step should be.

We therefore propose, as a substitute, *incremental* invariant-strengthening, which mitigates this problem. It is based on the following notion.

Definition 4 (Conditionally Stable Predicate). *A state predicate r is* conditionally stable *with respect to a state predicate r_1 if $\overset{\#}{\rightarrow}(r_1 \wedge r) \Rightarrow r$*

Conditional stability is easier to prove than stability because it is weaker.

Incremental invariant-strengthening starts just like plain invariant strengthening – proving $l \Rightarrow r$ and attempting to prove $\overset{\#}{\rightarrow}(r) \Rightarrow r$. But, at the first incremental-strengthening step, the user does not attempt to prove that $r \wedge r_1$ is stable; rather, one proves that r is *conditionally* stable with respect to r_1 – a weaker and simpler statement that unconditional stability – and then one *attempts to prove that r_1 is an invariant from l, by recursively using incremental invariant-strengthening.* Hence, at each step, the user deals with simpler predicates than in plain invariant-strengthening, and failures to prove the stability of simpler predicates is easier to exploit for figuring out the next relevant incremental strengthening step. This makes a lot of difference in practice.

We now formally define incremental invariant-strengthening as a proof system. We define the relation $\Vdash$ between transition systems $\mathcal{S}$ and invariance formulas $l \Rightarrow \Box r$ to be the smallest relation defined by the proof system in Figure 5. The thus-defined incremental invariant-strengthening process is sound:

Theorem 3 (Soundness for Incremental Invariant Strengthening). *If $\mathcal{S} \Vdash l \Rightarrow \Box r$ then $\mathcal{S} \vDash l \Rightarrow \Box r$.*

Proof. For the rules [End] and [Cnt] as shown in Figure 5, we say that [End] *is applied with parameters l and r* when l and r are the state predicates occurring in the rule's instance. Similarly, we say that [Cnt] *is applied with parameters l, r, and r'* when l, r, and r' are the predicates occurring in the rule's instance.

We first define the partial function $\bigwedge : S^{\#} \to S^{\#}$ (which depends on the assumed transition system $(\mathcal{S}, \mapsto)$) by induction on the definition of the proof of $\mathcal{S} \Vdash l \Rightarrow \Box r$ as follows:

- for applications of [End] with parameters l and r, $\bigwedge r \triangleq r$;

- for applications of [Cnt] with parameters l, r, and r', $\bigwedge r \triangleq r \wedge (\bigwedge r')$.

Then, we prove by induction on the proof of $\mathcal{S} \Vdash l \Rightarrow \Box r$ that $l \Rightarrow (\bigwedge r)$, $(\bigwedge r) \Rightarrow r$ and $(\overset{\#}{\rightarrow}(\bigwedge r)) \Rightarrow \bigwedge r$ (in particular, $\bigwedge r$ is initialised in l and stable):

- for the base case, i.e., the proof consists in one application of [End] with parameters l and r: the three implications to prove result from $\bigwedge r \triangleq r$ and from the conditions $l \Rightarrow r$ and $(\overset{\#}{\rightarrow}(r)) \Rightarrow r$ for applying [End];
- for the inductive step, i.e., an application of [Cnt] with parameters l, r, r' followed by a proof of $\mathcal{S} \Vdash l \Rightarrow \Box r'$: we have $\bigwedge r \triangleq r \wedge (\bigwedge r')$ and the inductive hypotheses are $l \Rightarrow (\bigwedge r')$, $(\bigwedge r') \Rightarrow r'$, $(\overset{\#}{\rightarrow}(\bigwedge r')) \Rightarrow (\bigwedge r')$:
  - the first implication $l \Rightarrow (\bigwedge r)$ results from $l \Rightarrow (\bigwedge r')$, $\bigwedge r \triangleq r \wedge (\bigwedge r')$, and the condition $l \Rightarrow r$ for applying the [Cnt] rule;
  - the second implication $\bigwedge r \Rightarrow r$ results from defining $\bigwedge r \triangleq r \wedge (\bigwedge r')$:
  - the third implication $(\overset{\#}{\rightarrow}(\bigwedge r)) \Rightarrow \bigwedge r$ is obtained as follows. We have $\overset{\#}{\rightarrow}(r \wedge (\bigwedge r')) \Rightarrow (\overset{\#}{\rightarrow}(r \wedge r'))$ by monotonicity and using the inductive hypothesis $\bigwedge r' \Rightarrow r'$. Moreover, $(\overset{\#}{\rightarrow}(r \wedge r')) \Rightarrow r$ from the condition for applying [Cnt]. By transitivity, $\overset{\#}{\rightarrow}(r \wedge (\bigwedge r')) \Rightarrow r$, i.e, (†) $(\overset{\#}{\rightarrow}(\bigwedge r)) \Rightarrow r$ by the definition of $\bigwedge r$. Next, $\overset{\#}{\rightarrow}(r \wedge (\bigwedge r')) \Rightarrow \overset{\#}{\rightarrow}(\bigwedge r')$ by monotonicity, and $(\overset{\#}{\rightarrow}(\bigwedge r')) \Rightarrow (\bigwedge r')$ is an inductive hypothesis, hence, $(\overset{\#}{\rightarrow}(\bigwedge r)) = \overset{\#}{\rightarrow}(r \wedge (\bigwedge r')) \Rightarrow r(\bigwedge r')$, which, combined with (†), settles the third implication and completes the proof by induction.

Finally, from the above-established $l \Rightarrow (\bigwedge r)$ and $(\overset{\#}{\rightarrow}(\bigwedge r)) \Rightarrow \bigwedge r$ we obtain by Lemma 4 that $\bigwedge r$ is an invariant from l, i.e., by Definition 3, for each state s reachable from l, $(\bigwedge r)(s)$. But we also established $(\bigwedge r) \Rightarrow r$, hence, for each state s reachable from l, $r(s)$ holds. By Definition 3, r is an invariant from l. □

Theorem 4 (Completeness for Incremental Invariant Strengthening). *If $\mathcal{S} \vDash l \Rightarrow \Box r$ then $\mathcal{S} \Vdash l \Rightarrow \Box r$.*

Proof. We first formalise the set of states R reachable from l (encountered in Definition 3) as a state predicate. Let $\mathcal{R} = \lambda s.\ l \overset{*}{\mapsto} s$, where $\overset{*}{\mapsto}$ is the smallest subset of $S^{\#} \times S$ defined by the rules

$$[\mathsf{Init}]\ \frac{}{l \overset{*}{\mapsto} s}\ \textit{if}\ l(s)$$

$$[\mathsf{Trans}]\ \frac{l \overset{*}{\mapsto} s}{l \overset{*}{\mapsto} s'}\ \textit{if}\ s \mapsto s'$$

In Definition 3, invariants from l can be equivalently defined as state predicates r satisfying $\mathcal{R} \Rightarrow r$. We first note that $\overset{\#}{\rightarrow}(\mathcal{R}) \Rightarrow \mathcal{R}$: assume any state s' such

that $(\overset{\#}{\rightarrow}(\mathcal{R}))(s')$, then by definition of the state transition function $\overset{\#}{\rightarrow} : S^{\#} \rightarrow S^{\#}$, there is a state s such that $\mathcal{R}(s)$ and $s \mapsto s'$. From the definition of $\mathcal{R}$ we obtain $l \overset{*}{\mapsto} s$ and using the rule [Trans], $l \overset{*}{\mapsto} s'$, i.e., $\mathcal{R}(s')$: hence, $\overset{\#}{\rightarrow}(\mathcal{R}) \Rightarrow \mathcal{R}$ is proved.

We also note that $l \Rightarrow \mathcal{R}$ holds: indeed, for any state s, if $l(s)$ then, using [Init], $l \overset{*}{\mapsto} s$, which implies $\mathcal{R}(s)$ by definition of $\mathcal{R}$.

Consider now any state predicate r that is invariant from l. We construct the following proof of $\mathcal{S} \Vdash l \Rightarrow \Box r$ using the proof system represented in Figure 5:

- apply [Cnt] with parameters l, r and $\mathcal{R}$: the rule can indeed be applied, since, per the above, $l \Rightarrow \mathcal{R}$ and $\mathcal{R} \Rightarrow r$, hence, the first condition $l \Rightarrow r$ holds, and the second condition$((\overset{\#}{\rightarrow}(\mathcal{R} \wedge r))) \Rightarrow r$ holds because $((\overset{\#}{\rightarrow}(\mathcal{R} \wedge r))) \Rightarrow \overset{\#}{\rightarrow}(\mathcal{R})$ (monotonicity), $\overset{\#}{\rightarrow}(\mathcal{R}) \Rightarrow \mathcal{R}$ (proved above) and $\mathcal{R} \Rightarrow r$;
- apply [End] with parameters l and $\mathcal{R}$: the rule can indeed be applied, since its conditions $l \Rightarrow \mathcal{R}$ and $\overset{\#}{\rightarrow}(\mathcal{R}) \Rightarrow \mathcal{R}$ have been established above.

Thus, from $\mathcal{S} \vDash l \Rightarrow \Box r$ we established $\mathcal{S} \Vdash l \Rightarrow \Box r$, which concludes the proof. □

A natural question that arises is whether *infinite*, i.e. coinductive incremental invariant-strengthening adds any expressiveness to the finite, inductive one defined above. (We noted that that was the case for the proof system for RL formulas.) The answer is, in theory, no: the finite version is already complete. However, a coinductive version could deal with circular situations where, i.e., r_1 is conditionally stable w.r.t. r_2 and r_2 is conditionally stable w.r.t. r_1. In this case, the inductive version needs to revert to "plain" invariant strengthening, i.e., proving that the conjunction $r_1 \wedge r_2$ is (unconditionally) stable, whereas a coinductive version does not have to do this. This question, and, of course, proving the soundness such a coinductive proof system, are left for future work.

Like for the RL proof system, the completeness and soundness of the proof system for invariance allows us to add new proof rules. For example, one such proof rule reduces $\mathcal{S} \Vdash l \Rightarrow \Box(r_1 \wedge r_2)$ to $\mathcal{S} \Vdash l \Rightarrow \Box r_1$ and $\mathcal{S} \Vdash l \Rightarrow \Box r_2$. This rule is sound because the corresponding implication with $\vDash$ instead of $\Vdash$ holds: i.e., $\mathcal{S} \vDash l \Rightarrow \Box r_1$ and $\mathcal{S} \vDash l \Rightarrow \Box r_2$ imply $\mathcal{S} \vDash l \Rightarrow \Box(r_1 \wedge r_2)$; which holds since it amounts to proving that $\mathcal{R} \Rightarrow r_1$ and $\mathcal{R} \Rightarrow r_2$ imply $\mathcal{R} \Rightarrow (r_1 \wedge r_2)$ with $\mathcal{R}$ defined as in the proof of Theorem 4. The predicate $\mathcal{R}$ also enables us to prove:

Theorem 5 (Reducing Reachability to Invariance). *$\mathcal{S} \vDash l \Rightarrow \Diamond r$ whenever there exists a state predicate $\mathcal{H}$ such that $\mathcal{S} \vDash l \Rightarrow \Box \mathcal{H}$ and $(\mathcal{H} \wedge final) \Rightarrow r$.*

Proof. From $l \Rightarrow \Box \mathcal{H}$ we obtain $\mathcal{R} \Rightarrow \mathcal{H}$, with $\mathcal{R}$ defined as in the proof of Theorem 4. In that proof we also showed $l \Rightarrow \mathcal{R}$ and $\overset{\#}{\rightarrow}(\mathcal{R}) \Rightarrow \mathcal{R}$. Since $(\mathcal{H} \wedge final) \Rightarrow r$, we also obtain $(\mathcal{R} \wedge final) \Rightarrow r$ by monotonicity of implication. Hence, using Lemma 3 with $\mathcal{J} := \mathcal{R}$ we obtain the desired conclusion $\mathcal{S} \vDash l \Rightarrow \Diamond r$. □

We use Coq implementations of the thus-defined incremental invariant-strengthening technique, and of the above-established reduction of reachability to invariance, for proving the functional correctness of our hypervisor model.

6. Hypervisor Model Example

6.1. General Description

We now describe our verification example: a security hypervisor for machine code. The main idea is that the supervisor "scans" all machine-code instructions before letting them be executed by a processor in kernel mode. The hypervisor then analyses the opcode of the currently scanned instruction, to determine whether the instruction is *normal* or *special*. Technically speaking, this is done by looking up the opcode in a table containing all opcodes and their normal/special status. 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 security risk when executed in kernel mode. This includes jumps to addresses that are computed at runtime, because their destination could be in the kernel-memory space, as well as some kernel-specific instructions, e.g., modifying internal data structures of the kernel for memory management.

If an instruction is normal then the hypervisor safely passes it on to the processor for execution. However, if an instruction is special, the hypervisor takes appropriate actions such as emulating the instruction in a safe and controlled way, or, in extreme cases, blocking any further code execution.

The main *functional correctness* properties of the hypervisor are that *all instructions passed on to the processor are normal*, and *when the code execution terminates, its effect is the same as that obtained by running it un-hypervised*. 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 it should increase the global execution time as little as possible. This excludes, for example, machine-code emulation of normal instructions, since the emulation is several magnitude-orders slower than hardware execution in a processor. Another unrealistic design is to scan, analyse, and execute (or emulate) instructions one by one: the alternation between analysis and execution/emulation is a major source of execution-time overhead, because it involves costly operations of saving/restoring software images of the processor's state.

Thus, in order to avoid being too slow, the hypervisor must analyse as many instructions as possible before letting them be executed by the processor.

A general graphical depiction of the hypervisor is shown in Figure 6. The locations correspond to several *modes* in which the hypervisor (plus the hypervised system) may be. In *hyper* mode the hypervisor scans and analyses instructions. If an instruction is *normal* the hypervisor accepts it and goes on to the next instruction, which in most cases is just the actual next-in-sequence instruction. Except, of course, for *jump* instructions, which typically branch at different addresses than that of the next-in-sequence one. Since the hypervisor only performs a static analysis and some limited emulation, it has, in general, no way of knowing which is the next instruction that it should scan after a jump.

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

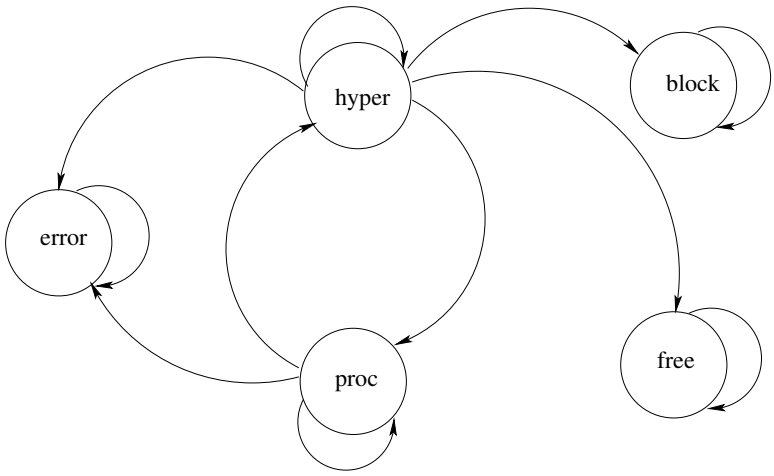

Figure 6: General structure of hypervisor.

current sequence of scanned and analysed instructions (i.e., the ones deemed normal, and the new trap instruction) is "flushed" to the processor for execution. In our state machine in Figure 6 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, based on which the hypervisor "knows" what action to take; e.g., either continue with scanning the instruction after the jump, at an address that is now known and considered safe, or blocking code execution if the address is considered unsafe. Thus, in terms of the state machine in Figure 6, the system goes back to *hyper*, or goes to *block*, after having restored the *jump* instruction in order to avoid code alterations.

The same mechanism is used when any other special instruction is encountered by the hypervisor: it is replaced by a *trap*, the system goes to *proc* mode for executing the current list of scanned and analysed instructions, and when *trap* is executed, appropriate action is taken by the hypervisor: execution is either blocked, or the special instruction is safely emulated and the system switches back to *hyper* mode, after having restored the special instruction in its place.

As already stated previously these "switches" from scanning/analysis to execution generate much execution overhead and are avoided whenever possible. 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 among the ones generated by normal, possibly conditional *jump* instructions, whose destinations are statically known.

The idea is to save in the hypervisor's state the *list of instructions already scanned and found to be normal' during the current presence in the "hyper" mode*. 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-scanned instruction list, then the scanning can safely continue at the other address since a second scanning of a "safe" instruction sequence is useless. Moreover, if both addresses 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 scanned 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-scanned instruction list. Only in the remaining cases (all jumps go outside the list) is mode switching via trap-and-execution required for discovering the next instruction to scan.

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 its instructions needs, of course, to be scanned and analysed; but when the conditional jump of the loop's compiled code is encountered a second time, the hypervisor need not "solve" the condition: all instructions in the loop body have already been scanned and deemed normal, thus, the processor will safely execute them; the scanning continues at the instruction after the loop body.

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

6.2. Coq Model of the Hypervisor

The transition-system model of the hypervisor has the general structure of the state machine shown in Figure 6. 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 list of instructions 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 of our hypervisor model we describe some additional artefacts.

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`[5]), 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 code that it hypervises.

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 depend on `procState` (e.g., on register values) is not determined statically but dynamically.

The static control flow is used during the scannning/analysis phases since, for efficiency reasons, the hypervisor only emulates a small fraction of the analysed code; 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 intention is that the natural numbers `n` and `m`, when present, are indices in the list of instructions that appears as the argument of the (parameter) function modelling the static control flow:

```
nxt :  list Ins → nat → Next
```

Specifically, given a list of instructions and a natural-number position, `nxt` returns the next statically-known address(es) of the next instruction(s) for the instruction at the given position in the given list. 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
```

[5] Option types will also be used by other artefacts of our Coq model.

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 for the instruction in question have to coincide.

Code instrumentation. When the hypervisor encounters a special instruction, including jumps 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 of branching execution in the hypervisor, which takes appropriate action (e.g., safely emulating 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 axiomatically 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.

Transition relation. The transition relation `trans:State→State→Prop` is defined using the `Inductive` keyword, just like the one shown in Section 2 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.

6.3. Coq Proof of the 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 dangerous) 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 un-hypervised. The first property must hold at all times, since special instructions may occur at any time; by contrast, the second property needs only to hold at the end: while a special instruction is being emulated, the property may not hold.[6]

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 scanned/analysed before 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. The property is stated as the following theorem (where `Valid` is the Coq version of the notation "$\mathcal{S} \vDash \ldots$" used so far):

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

Here, `fun` defines a predicate, which holds for the states whose relevant components: `mode`, instruction pointer `lo`, and `code` (extracted from states using `match`) 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 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 a `halt` or a `trap 0` replacing 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 defined 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

[6] We note that the second propety could also be expressed as an invariant, but that would have to include additional information, i.e., for precisely specifying the special contexts when the invariant does not hold. By contrast, a partial-correctness property does not have to specify those contexts; it is therefore more compact, and, ultimately, more to the point.

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 invariance property. For invariance properties we use a Coq implementation of the incremental invariant-strengthening technique presented in Section 5. We used 23 predicates to attain stability under the transition relation.

Proving the partial-correctness property. We apply a Coq implementation of Theorem 5 that reduces RL formula validity to the discovery of a strong-enough invariant. For proving the RL formula of interest, an invariant consisting of a conjunction of 30 predicates was found. Fortunately we were able to reuse the 23 predicates required for proving the invariance property; hence, verification of the partial-correctness property was an incremental effort over that of the invariance property, using the same technique of incremental invariant-strengthening.

This concludes the presentation of the hypervisor case study.

7. 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, coinductive proof system for RL in this setting, for which we prove soundness and completeness. While theoretical in nature, the completeness result also has a practical value as it suggests a strategy for the proof system that is able to deal with all valid RL formulas over a given transition system. The Coq mechanisation of these results provides us with a Coq-certified interactive prover for RL. The reduction of partial correctness to invariance, and an incremental approach for proving invariants, also formalised as a sound and complete proof system, were helpful in enabling us to complete a nontrivial example of a security hypervisor model within reasonable time and effort limits.

The main line of future work is exploiting our RL proof system in more general ways than the strategy of reducing each formula's proof to that of one (typically large) 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. The inclusion of additional proof rules in our RL proof system, enabled by the soundness and completeness results as illustrated in the paper, is a first step towards such more general proofs. We are also planning to investigate whether our proof systems for reachability $\Rightarrow\Diamond$ and for invariance $\Rightarrow\Box$, two modalities which correspond to homonymous modalities of LTL on finite paths, can be combined into a system for proving, e.g., LTL formulas with nested modalities.

Acknowledgements. We thank the anonymous reviewers of earlier versions of this paper for their useful hints and suggestions, especially the suggestion to use coinduction, and David Nowak for his lessons on coinduction in Coq.

[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] D. Cousineau and D. Doligez and L. Lamport and S. Merz and D. Ricketts and H. Vanzetto. TLA$^+$ Proofs. In *FM 2012*, Springer LNCS 7436, pages 147–154.

[4] The Isabelle/HOL proof assistant. `https://isabelle.in.tum.de`.

[5] The Coq proof assistant. `http://coq.inria.fr`.

[6] G. Roşu and A. Ştefănescu. Towards a unified theory of operational and axiomatic semantics. In *ICALP 2012*, Springer LNCS 7392, pages 351–363.

[7] G. Roşu and A. Ştefănescu. Checking reachability using matching logic. In *OOPSLA 2012*, ACM, pages 555–574.

[8] G. Roşu, A. Ştefănescu, Ş. Ciobâcă, and B. Moore. One-path reachability logic. In *LICS 2013*, IEEE, pages 358–367.

[9] A. Ştefănescu, Ş. Ciobâcă, R. Mereuţă, B. Moore, T. F. Şerbănuţă, and G. Roşu. All-paths reachability logic. In *RTA 2014*, Springer LNCS 8560, pages 425–440.

[10] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. Semantics-based program verifiers for all languages. In *OOPSLA 2016*, ACM, pages 74–91.

[11] 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)*.

[12] V. Rusu, G. Grimaud, and M. Hauspie. Proving Partial-Correctness and Invariance Properties of Transition-System Models. In *TASE 2018*, IEEE, pages 60–67. Available at `https://hal.inria.fr/hal-01816798`.

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

[14] D. Lucanu, V. Rusu, and A. Arusoaie. A generic framework for symbolic execution: A coinductive approach. *J. Symb. Comput.*, 80:125–163, 2017.

[15] Ş. Ciobâcă and D. Lucanu. A Coinductive Approach to Proving Reachability Properties in Logically Constrained Term Rewriting Systems. In *IJCAR 2018*, Springer LNCS 10900, pages 295–311.

[16] B. Moore, L. Peña, and G. Roşu. Program Verification by Coinduction. in *ESOP' 2018*, Springer LNCS 10801, pages 589–618.

[17] A. Arusoaie, D. Nowak, D. Lucanu, and V. Rusu, A Certified Procedure for RL Verification. In *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNACS'17)*, IEEE, 2017.

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

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

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

[21] A. 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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 1530

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

[29] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and S. Winwood. SeL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 6(53):107-115, 2010.

[30] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjberg, and D. Costanzo. CertiKOS: An Extenisble Architecture for Building Certified Concurrent OS Kernels. *USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 653-669.

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

[32] A. Chlipala. Certified programs with dependent types. Available at `http://adam.chlipala.net/cpdt`.

[33] E. Gimenez. A Calculus of Infinite Constructions and its application to the verification of communicating systems. PhD thesis, Ecole Normale Superieure de Lyon, 1996.

[34] G. De Giacomo and M. Y.Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI 2013*, pages 854-860.