RTA'14, LNCS 8560, pp 425-440. 2014

All-Path Reachability Logic [⋆]

Andrei Ştefănescu[1], Ştefan Ciobâcă[2], Radu Mereuta[1,2],
Brandon M. Moore[1], Traian Florin Şerbănuţă[3], and Grigore Roşu[1,2]

[1] University of Illinois at Urbana-Champaign, USA
{stefane1,radum,bmoore,grosu}@illinois.edu
[2] University "Alexandru Ioan Cuza", Romania, stefan.ciobaca@info.uaic.ro
[3] University of Bucharest, Romania, traian.serbanuta@fmi.unibuc.ro

Abstract. This paper presents a language-independent proof system for reachability properties of programs written in non-deterministic (e.g. concurrent) languages, referred to as *all-path reachability logic*. It derives partial-correctness properties with all-path semantics (a state satisfying a given precondition reaches states satisfying a given postcondition on all terminating execution paths). The proof system takes as axioms any unconditional operational semantics, and is sound (partially correct) and (relatively) complete, independent of the object language; the soundness has also been mechanized (Coq). This approach is implemented in a tool for semantics-based verification as part of the $\mathbb{K}$ framework.

1 Introduction

Operational semantics are easy to define and understand. Giving a language an operational semantics can be regarded as "implementing" a formal interpreter. Operational semantics require little formal training, scale up well and, being executable, can be tested. Thus, operational semantics are typically used as trusted reference models for the defined languages. Despite these advantages, operational semantics are rarely used directly for program verification (i.e. verifying properties of a given program, rather than performing meta-reasoning about a given language), because such proofs tend to be low-level and tedious, as they involve formalizing and working directly with the corresponding transition system. Hoare or dynamic logics allow higher level reasoning at the cost of (re)defining the language as a set of abstract proof rules, which are harder to understand and trust. The state-of-the-art in mechanical program verification is to develop and prove such language-specific proof systems sound w.r.t to a trusted operational semantics [1–3], but that needs to be done for each language separately.

Defining more semantics for the same language and proving the soundness of one semantics in terms of another are highly uneconomical tasks when real programming languages are concerned, often taking several years to complete. Ideally, we would like to have only one semantics for a language, together with a generic theory and a set of generic tools and techniques allowing us to get all the benefits of any other semantics without paying the price of defining other semantics. Recent work [4–7] shows this *is possible*, by proposing a *language-independent proof system* which derives program properties directly from an operational semantics, *at the same proof granularity and compositionality* as a language-specific axiomatic semantics. Specifically, it introduces *(one-path) reachability rules*, which generalize both operational semantics reduction

[⋆] Full version of this paper, with proofs, available at http://hdl.handle.net/2142/46296.

rules and Hoare triples, and give a proof system which derives new reachability rules (program properties) from a set of given reachability rules (the language semantics).

However, the existing proof system has a major limitation: it only derives reachability rules with a *one-path* semantics, that is, it guarantees a program property holds on one but not necessarily all execution paths, which suffices for deterministic languages but not for non-deterministic (concurrent) languages. We here remove this limitation, proposing the first generic all-path reachability proof system for program verification.

Using *matching logic* [8] as a configuration specification formalism (Section 2), where a pattern φ specifies all program configurations that match it, we first introduce the novel notion of an *all-path reachability rule* $\varphi \Rightarrow^{\forall} \varphi'$ (Section 3), where φ and φ' are matching logic patterns. Rule $\varphi \Rightarrow^{\forall} \varphi'$ is valid iff any program configuration satisfying φ reaches, on any complete execution path, some configuration satisfying φ'. This subsumes partial-correctness in non-deterministic languages. We then present a proof system for deriving an all-path reachability rule $\varphi \Rightarrow^{\forall} \varphi'$ from a set $\mathcal{S}$ of semantics rules (Section 4). $\mathcal{S}$ consists of reduction rules $\varphi_l \Rightarrow^{\exists} \varphi_r$, where φ_l and φ_r are simple patterns as encountered in operational semantics (Section 6), which can be non-deterministic. The proof system derives more general sequents "$\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^{\forall} \varphi'$", with $\mathcal{A}$ and C two sets of reachability rules. Intuitively, $\mathcal{A}$'s rules (*axioms*) are already established valid, and thus can be immediately used. Those in C (*circularities*) are only claimed valid, and can be used only after taking execution steps based on the rules in $\mathcal{S}$ or $\mathcal{A}$. The most important proof rules are

Step :

$$\frac{\begin{array}{l}\models \varphi \rightarrow \bigvee_{\varphi_l \Rightarrow^{\exists} \varphi_r \,\in\, \mathcal{S}} \exists \mathit{FreeVars}(\varphi_l)\, \varphi_l \\ \models \exists c\, (\varphi[c/\square] \wedge \varphi_l[c/\square]) \wedge \varphi_r \rightarrow \varphi' \qquad \text{for each } \varphi_l \Rightarrow^{\exists} \varphi_r \,\in\, \mathcal{S}\end{array}}{\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^{\forall} \varphi'}$$

Circularity :

$$\frac{\mathcal{S}, \mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow^{\forall} \varphi'\}} \varphi \Rightarrow^{\forall} \varphi'}{\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^{\forall} \varphi'}$$

Step is the key proof rule which deals with non-determinism: it derives a sequent where φ reaches φ' in one step on all paths. The first premise ensures that any configuration satisfying φ has successors, the second that all successors satisfy φ' ($\square$ is the configuration placeholder). Circularity adds the current goal to C at any point in a proof, and generalizes language-independently the various language-specific axiomatic semantics invariant rules (this form was introduced in [4]).

We illustrate on examples how our proof system enables state exploration (similar to symbolic model-checking), and verification of program properties (Section 6). We show that our proof system is sound and relatively complete (Section 5). We describe our implementation of the proof system as past of the $\mathbb{K}$ framework [9] (Section 7).

Contributions This paper makes the following specific contributions:

1. A language-independent proof system for deriving all-path reachability properties, with proofs of its soundness and relative completeness; the soundness result has also been mechanized in Coq, to serve as a foundation for certifiable verification.
2. An implementation of it as part of the $\mathbb{K}$ framework.

2 Matching Logic

Here we briefly recall matching logic [8], which is a logic designed for specifying and reasoning about arbitrary program and system configurations. A matching logic formula, called a *pattern*, is a first-order logic (FOL) formula with special predicates, called basic patterns. A *basic pattern* is a configuration term with variables. Intuitively, a pattern specifies both structural and logical constraints: a configuration satisfies the pattern iff it matches the structure (basic patterns) and satisfies the constraints.

Matching logic is parametric in a signature and a model of configurations, making it a prime candidate for expressing state properties in a language-independent verification framework. The configuration signature can be as simple as that of IMP (Fig. 3), or as complex as that of the C language [10] (with more than 70 semantic components).

We use basic concepts from multi-sorted first-order logic. Given a *signature* Σ which specifies the sorts and arities of the function symbols (constructors or operators) used in configurations, let $T_\Sigma(\mathit{Var})$ denote the *free Σ-algebra* of terms with variables in Var. $T_{\Sigma,s}(\mathit{Var})$ is the set of Σ-terms of sort s. A valuation $\rho : \mathit{Var} \to \mathcal{T}$ with $\mathcal{T}$ a Σ-algebra extends uniquely to a (homonymous) *Σ-algebra morphism* $\rho : T_\Sigma(\mathit{Var}) \to \mathcal{T}$. Many mathematical structures needed for language semantics have been defined as Σ-algebras, including: boolean algebras, natural/integer/rational numbers, lists, sets, bags (or multisets), maps (e.g., for states, heaps), trees, queues, stacks, etc.

Let us fix the following: (1) an algebraic signature Σ, associated to some desired configuration syntax, with a distinguished sort *Cfg*, (2) a sort-wise infinite set *Var* of variables, and (3) a Σ-algebra $\mathcal{T}$, the *configuration model*, which may but need not be a term algebra. As usual, $\mathcal{T}_{\mathit{Cfg}}$ denotes the elements of $\mathcal{T}$ of sort *Cfg*, called *configurations*.

Definition 1. *[8] A matching logic formula, or a* ***pattern****, is a first-order logic (FOL) formula which additionally allows terms in $T_{\Sigma,\mathit{Cfg}}(\mathit{Var})$, called* ***basic patterns****, as predicates. A pattern is* ***structureless*** *if it contains no basic patterns.*

We define satisfaction $(\gamma,\rho) \models \varphi$ over configurations $\gamma \in \mathcal{T}_{\mathit{Cfg}}$, valuations $\rho : \mathit{Var} \to \mathcal{T}$ and patterns φ as follows (among the FOL constructs, we only show $\exists$):

$(\gamma,\rho) \models \exists X\ \varphi$ *iff* $(\gamma,\rho') \models \varphi$ *for some* $\rho' : \mathit{Var} \to \mathcal{T}$ *with* $\rho'(y) = \rho(y)$ *for all* $y \in \mathit{Var} \backslash X$

$(\gamma,\rho) \models \pi$ *iff* $\gamma = \rho(\pi)$ *where* $\pi \in T_{\Sigma,\mathit{Cfg}}(\mathit{Var})$

We write $\models \varphi$ when $(\gamma,\rho) \models \varphi$ for all $\gamma \in \mathcal{T}_{\mathit{Cfg}}$ and all $\rho : \mathit{Var} \to \mathcal{T}$.

A basic pattern π is satisfied by all the configurations γ that *match* it; in $(\gamma,\rho) \models \pi$ the ρ can be thought of as the "witness" of the matching, and can be further constrained in a pattern. For instance, the pattern from Section 6

$$\langle \texttt{x:=x+1} \parallel \texttt{x:=x+1},\ \texttt{x} \mapsto n \rangle \wedge (n = 0 \vee n = 1)$$

is matched by the configurations with code "`x:=x+1 || x:=x+1`" and state mapping program variable `x` into integer n with n being either 0 or 1. We use *italic* for mathematical variables in *Var* and `typewriter` for program variables (program variables are represented in matching logic as constants of sort *PVar*, see Section 6).

Next, we recall how matching logic formulae can be translated into FOL formulae, so that its satisfaction becomes FOL satisfaction in the model of configurations, $\mathcal{T}$. Then, we can use conventional theorem provers or proof assistants for pattern reasoning.

Definition 2. *[8] Let $\square$ be a fresh Cfg variable. For a pattern φ, let $\varphi^\square$ be the FOL formula formed from φ by replacing basic patterns $\pi \in \mathcal{T}_{\Sigma,\mathit{Cfg}}(\mathit{Var})$ with equalities $\square = \pi$. If $\rho : \mathit{Var} \to \mathcal{T}$ and $\gamma \in \mathcal{T}_{\mathit{Cfg}}$ then let the valuation $\rho^\gamma : \mathit{Var} \cup \{\square\} \to \mathcal{T}$ be such that $\rho^\gamma(x) = \rho(x)$ for $x \in \mathit{Var}$ and $\rho^\gamma(\square) = \gamma$.*

With the notation in Definition 2, $(\gamma,\rho) \models \varphi$ iff $\rho^\gamma \models \varphi^\Box$, and $\models \varphi$ iff $\mathcal{T} \models \varphi^\Box$. Thus, matching logic is a methodological fragment of the FOL theory of $\mathcal{T}$. We drop $\Box$ from $\varphi^\Box$ when it is clear in context that we mean the FOL formula instead of the matching logic pattern. It is often technically convenient to eliminate $\Box$ from φ, by replacing $\Box$ with a *Cfg* variable c and using $\varphi[c/\Box]$ instead of φ. We use the FOL representation in the Step proof rule in Fig. 1, and to establish relative completeness in Section 5.

3 Specifying Reachability

In this section we define one-path and all-path reachability. We begin by recalling some matching logic reachability [6] notions that we need for specifying reachability.

Definition 3. *[6] A (one-path)* ***reachability rule*** *is a pair $\varphi \Rightarrow^\exists \varphi'$, where φ and φ' are patterns (which can have free variables). Rule $\varphi \Rightarrow^\exists \varphi'$ is* ***weakly well-defined*** *iff for any $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$ with $(\gamma,\rho) \models \varphi$, there exists $\gamma' \in \mathcal{T}_{Cfg}$ with $(\gamma',\rho) \models \varphi'$. A* ***reachability system*** *is a set of reachability rules. Reachability system $\mathcal{S}$ is* ***weakly well-defined*** *iff each rule is weakly well-defined. $\mathcal{S}$ induces a* ***transition system*** *$(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ on the configuration model: $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ for $\gamma, \gamma' \in \mathcal{T}_{Cfg}$ iff there is some rule $\varphi \Rightarrow^\exists \varphi'$ in $\mathcal{S}$ and some valuation $\rho : Var \to \mathcal{T}$ with $(\gamma,\rho) \models \varphi$ and $(\gamma',\rho) \models \varphi'$. A* ***$\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path*** *is a finite sequence $\gamma_0 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} ... \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_n$ with $\gamma_0,...,\gamma_n \in \mathcal{T}_{Cfg}$. A $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path is* ***complete*** *iff it is not a strict prefix of any other $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path.*

We assume an operational semantics is a set of (unconditional) reduction rules "$l \Rightarrow^\exists r$ if b", where $l, r \in T_{\Sigma,Cfg}(Var)$ are program configurations with variables and $b \in T_{\Sigma,Bool}(Var)$ is a condition constraining the variables of l, r. Styles of operational semantics using only such (unconditional) rules include evaluation contexts [11], the chemical abstract machine [12] and $\mathbb{K}$ [9] (see Section 6 for an evaluation contexts semantics). Several large languages have been given semantics in such styles, including C [10] (about 1200 rules) and R5RS Scheme [13]. The reachability proof system works with any set of rules of this form, being agnostic to the particular style of semantics.

Such a rule "$l \Rightarrow^\exists r$ if b" states that a ground configuration γ which is an instance of l and satisfies the condition b reduces to an instance γ' of r. Matching logic can express terms with constraints: $l \wedge b$ is satisfied by exactly the γ above. Thus, we can regard such a semantics as a particular weakly well-defined reachability system $\mathcal{S}$ with rules of the form "$l \wedge b \Rightarrow^\exists r$". The weakly well-defined condition on $\mathcal{S}$ guarantees that if γ matches the left-hand-side of a rule in $\mathcal{S}$, then the respective rule induces an outgoing transition from γ. The transition system induced by $\mathcal{S}$ describes precisely the behavior of any program in any given state. In [4–6] we show that reachability rules capture one-path reachability properties and Hoare triples for deterministic languages.

Formally, let us fix an operational semantics given as a reachability system $\mathcal{S}$. Then, we can specify reachability in the transition system induced by $\mathcal{S}$

Definition 4. *An* ***all-path reachability rule*** *is a pair $\varphi \Rightarrow^\forall \varphi'$ of patterns φ and φ'.*

An all-path reachability rule $\varphi \Rightarrow^\forall \varphi'$ is satisfied, $\mathcal{S} \models \varphi \Rightarrow^\forall \varphi'$, iff for all complete $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-paths τ starting with $\gamma \in \mathcal{T}_{Cfg}$ and for all $\rho : Var \to \mathcal{T}$ such that $(\gamma,\rho) \models \varphi$, there exists some $\gamma' \in \tau$ such that $(\gamma',\rho) \models \varphi'$.

A one-path reachability rule $\varphi \Rightarrow^\exists \varphi'$ is satisfied, $\mathcal{S} \models \varphi \Rightarrow^\exists \varphi'$, iff for all $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$ such that $(\gamma,\rho) \models \varphi$, there is either a $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path from γ to some γ' such that $(\gamma',\rho) \models \varphi'$, or there is a diverging execution $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_2 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \cdots$ from γ.

Step :

$$\frac{\begin{array}{l}\models \varphi \rightarrow \bigvee_{\varphi_l \Rightarrow^{\exists} \varphi_r \,\in\, \mathcal{S}} \exists \mathit{FreeVars}(\varphi_l)\ \varphi_l \\ \models \exists c\ (\varphi[c/\square] \wedge \varphi_l[c/\square]) \wedge \varphi_r \rightarrow \varphi' \qquad \text{for each } \ \varphi_l \Rightarrow^{\exists} \varphi_r \ \in\ \mathcal{S}\end{array}}{\mathcal{S}, \mathcal{A} \ \vdash_C \varphi \Rightarrow^{\forall} \varphi'}$$

Axiom :

$$\frac{\varphi \Rightarrow^{\forall} \varphi' \ \in \mathcal{A}}{\mathcal{S}, \mathcal{A} \ \vdash_C \varphi \Rightarrow^{\forall} \varphi'}$$

Reflexivity :

$$\frac{\cdot}{\mathcal{S}, \mathcal{A} \ \vdash \varphi \Rightarrow^{\forall} \varphi}$$

Transitivity :

$$\frac{\mathcal{S}, \mathcal{A} \ \vdash_C \varphi_1 \Rightarrow^{\forall} \varphi_2 \qquad \mathcal{S}, \mathcal{A} \cup C \ \vdash \varphi_2 \Rightarrow^{\forall} \varphi_3}{\mathcal{S}, \mathcal{A} \ \vdash_C \varphi_1 \Rightarrow^{\forall} \varphi_3}$$

Case Analysis :

$$\frac{\mathcal{S}, \mathcal{A} \ \vdash_C \varphi_1 \Rightarrow^{\forall} \varphi \qquad \mathcal{S}, \mathcal{A} \ \vdash_C \varphi_2 \Rightarrow^{\forall} \varphi}{\mathcal{S}, \mathcal{A} \ \vdash_C \varphi_1 \vee \varphi_2 \Rightarrow^{\forall} \varphi}$$

Abstraction :

$$\frac{\mathcal{S}, \mathcal{A} \ \vdash_C \varphi \Rightarrow^{\forall} \varphi' \qquad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{S}, \mathcal{A} \ \vdash_C \exists X\, \varphi \Rightarrow^{\forall} \varphi'}$$

Consequence :

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \qquad \mathcal{S}, \mathcal{A} \ \vdash_C \varphi_1' \Rightarrow^{\forall} \varphi_2' \qquad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{S}, \mathcal{A} \ \vdash_C \varphi_1 \Rightarrow^{\forall} \varphi_2}$$

Circularity :

$$\frac{\mathcal{S}, \mathcal{A} \ \vdash_{C \cup \{\varphi \Rightarrow^{\forall} \varphi'\}} \varphi \Rightarrow^{\forall} \varphi'}{\mathcal{S}, \mathcal{A} \ \vdash_C \varphi \Rightarrow^{\forall} \varphi'}$$

Fig. 1. Proof system for reachability. We make the standard assumption that the free variables of $\varphi_l \Rightarrow^{\exists} \varphi_r$ in the Step proof rule are fresh (e.g., disjoint from those of $\varphi \Rightarrow^{\forall} \varphi'$).

The racing increment example in Section 6 can be specified by

$$\langle \texttt{x:=x+1} \parallel \texttt{x:=x+1},\ \texttt{x}\mapsto m \rangle \Rightarrow^{\forall} \exists n\, (\langle \texttt{skip},\ \texttt{x}\mapsto n \rangle \wedge (n = m +_{Int} 1 \vee n = m +_{Int} 2)$$

which states that every terminating execution reaches a state where execution of both threads is complete and the value of `x` has increased by 1 or 2 (this code has a race).

A Hoare triple describes the resulting state after execution finishes, so it corresponds to a reachability rule where the right side contains no remaining code. However, all-path reachability rules are strictly more expressive than Hoare triples, as they can also specify intermediate configurations (the code in the right-hand-side need not be empty) Reachability rules provide a unified representation for both language semantics and program specifications: $\varphi \Rightarrow^{\exists} \varphi'$ for semantics and $\varphi \Rightarrow^{\forall} \varphi'$ for all-path reachability specifications. Note that, like Hoare triples, reachability rules can only specify properties of complete paths (that is, terminating execution paths). One can use existing Hoare logic techniques to break reasoning about a non-terminating program into reasoning about its terminating components.

4 Reachability Proof System

Fig. 1 shows our novel proof system for all-path reachability. The target language is given as a weakly well-defined reachability system $\mathcal{S}$. The soundness result (Thm. 1) guarantees that $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$ if $\mathcal{S} \ \vdash \varphi \Rightarrow^{\forall} \varphi'$ is derivable. Note that the proof system derives more general sequents of the form $\mathcal{S}, \mathcal{A} \ \vdash_C \varphi \Rightarrow^{\forall} \varphi'$, where $\mathcal{A}$ and C are sets of reachability rules. The rules in $\mathcal{A}$ are called *axioms* and rules in C are called *circularities*. If either $\mathcal{A}$ or C does not appear in a sequent, it means the respective set is empty: $\mathcal{S} \ \vdash_C \varphi \Rightarrow^{\forall} \varphi'$ is a shorthand for $\mathcal{S}, \emptyset \ \vdash_C \varphi \Rightarrow^{\forall} \varphi'$, and $\mathcal{S}, \mathcal{A} \ \vdash \varphi \Rightarrow^{\forall} \varphi'$ is a shorthand for $\mathcal{S}, \mathcal{A} \vdash_{\emptyset} \varphi \Rightarrow^{\forall} \varphi'$. Initially, both $\mathcal{A}$ and C are empty. Note that "$\rightarrow$" in Step and Consequence denotes implication.

The intuition is that the reachability rules in $\mathcal{A}$ can be assumed valid, while those in C have been postulated but not yet justified. After making progress from φ (at least one derivation by Step or by Axiom with the rules in $\mathcal{A}$), the rules in C become (coinductively) valid (can be used in derivations by Axiom). During the proof, circularities can be added to C via Circularity, flushed into $\mathcal{A}$ by Transitivity, and used via Axiom. The desired semantics for sequent $\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^{\forall} \varphi'$ (read "$\mathcal{S}$ with axioms $\mathcal{A}$ and circularities C proves $\varphi \Rightarrow^{\forall} \varphi'$") is: $\varphi \Rightarrow^{\forall} \varphi'$ holds if the rules in $\mathcal{A}$ hold and those in C hold after taking at least on step from φ in the transition system $(\Rightarrow^{\mathcal{T}}_{\mathcal{S}}, \mathcal{T})$, and if $C \neq \emptyset$ then φ reaches φ' after at least one step on all complete paths. As a consequence of this definition, any rule $\varphi \Rightarrow^{\forall} \varphi'$ derived by Circularity has the property that φ reaches φ' after at lest one step, due to Circularity having a prerequisite $\mathcal{S}, \mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow^{\forall} \varphi'\}} \varphi \Rightarrow^{\forall} \varphi'$ (with a non-empty set of circularities). We next discuss the proof rules.

Step derives a sequent where φ reaches φ' in one step on all paths. The first premise ensures any configuration matching φ matches the left-hand-side φ_l of some rule in $\mathcal{S}$ and thus, as $\mathcal{S}$ is weakly well-defined, can take a step. Formally, if $(\gamma, \rho) \models \varphi$, then there exists some rule $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ and some valuation ρ' of the free variables of φ_l such that $(\gamma, \rho') \models \varphi_l$, and thus γ has at least one $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-successor generated by the rule $\varphi_l \Rightarrow^{\exists} \varphi_r$. The second premise ensures that each $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-successor of a configuration matching φ matches φ'. Formally, if $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ and γ matches φ then there is some rule $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ and $\rho : \mathit{Var} \rightarrow \mathcal{T}$ such that $(\gamma, \rho) \models \varphi \wedge \varphi_l$ and $(\gamma', \rho) \models \varphi_r$; then the second part implies γ' matches φ'.

Designing a proof rule for deriving an execution step along all paths is non-trivial. For instance, one might expect Step to require as many premises as there are transitions going out of φ, as is the case for the examples presented later in this paper. However, that is not possible, as the number of successors of a configuration matching φ may be unbounded even if each matching configuration has a finite branching factor in the transition system. Step avoids this issue by requiring only one premise for each rule by which some configuration φ can take a step, even if that rule can be used to derive multiple transitions. To illustrate this situation, consider a language defined by $\mathcal{S} \equiv \{\langle n_1 \rangle \wedge n_1 >_{\mathit{Int}} n_2 \Rightarrow^{\exists} \langle n_2 \rangle\}$, with n_1 and n_2 non-negative integer variables. A configuration in this language is a singleton with a non-negative integer. Intuitively, a positive integer transits into a strictly smaller non-negative integer, in a non-deterministic way. The branching factor of a non-negative integer is its value. Then $\mathcal{S} \models \langle m \rangle \Rightarrow^{\forall} \langle 0 \rangle$. Deriving it reduces (by Circularity and other proof rules) to deriving $\langle m_1 \rangle \wedge m_1 >_{\mathit{Int}} 0 \Rightarrow^{\forall} \exists m_2\, (\langle m_2 \rangle \wedge m_1 >_{\mathit{Int}} m_2)$. The left-hand-side is matched by any positive integer, and thus its branching factor is infinity. Deriving this rule with Step requires only two premises, $\models (\langle m_1 \rangle \wedge m_1 >_{\mathit{Int}} 0) \rightarrow \exists n_1 n_2\, (\langle n_1 \rangle \wedge n_1 >_{\mathit{Int}} n_2)$ and $\models \exists c\, (c = \langle m_1 \rangle \wedge m_1 >_{\mathit{Int}} 0 \wedge c = \langle n_1 \rangle \wedge n_1 >_{\mathit{Int}} n_2) \wedge \langle n_2 \rangle \rightarrow \exists m_2\, (\langle m_2 \rangle \wedge m_1 >_{\mathit{Int}} m_2)$. A similar situation arises in real life for languages with thread pools of arbitrary size.

Axiom applies a trusted rule. Reflexivity and Transitivity capture the corresponding closure properties of the reachability relation. Reflexivity requires C to be empty to ensure that all-path rules derived with non-empty C take at least one step. Transitivity enables the circularities as axioms for the second premise, since if C is not empty, the first premise is guaranteed to take at least one step. Consequence, Case Analysis and Abstraction are adapted from Hoare logic. Ignoring circularities, these seven rules discussed so far constitute formal infrastructure for symbolic execution.

Circularity has a coinductive nature, allowing us to make new circularity claims. We typically make such claims for code with repetitive behaviors, such as loops, recursive functions, jumps, etc. If there is a derivation of the claim using itself as a circularity, then the claim holds. This would obviously be unsound if the new assumption was available immediately, but requiring progress (taking at least on step in the transition system $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$) before circularities can be used ensures that only diverging executions can correspond to endless invocation of a circularity.

One important aspect of concurrent program verification, which we do not address in this paper, is proof compositionality. Our focus here is limited to establishing a sound and complete language-independent proof system for all-path reachability rules, to serve as a foundation for further results and applications, and to discuss our current implementation of it. We only mention that we have already studied proof compositionality for earlier one-path variants of reachability logic [5], showing that there is a mechanical way to translate any Hoare logic proof derivation into a reachability proof of similar size and structure, but based entirely on the operational semantics of the language. The overall conclusion of our previous study, which we believe will carry over to all-path reachability, was that compositional reasoning can be achieved methodologically using our proof system, by proving and then using appropriate reachability rules as lemmas. However, note that this works only for theoretically well-behaved languages which enjoy a compositional semantics. For example, a language whose semantics assumes a bounded heap size, or which has constructs whose semantics involve the entire program, e.g., call/cc, will lack compositionality.

5 Soundness and Relative Completeness

Here we discuss the soundness and relative completeness of our proof system. Unlike the similar results for Hoare logics and dynamic logics, which are separately proved for each language taking into account the particularities of that language, we prove soundness and relative completeness once and for all languages.

Soundness states that a syntactically derivable sequent holds semantically. Because of the utmost importance of the result below, we have also mechanized its proof. Our complete Coq formalization can be found at `http://fsl.cs.illinois.edu/rl`.

Theorem 1 (Soundness). *If $\mathcal{S} \vdash \varphi \Rightarrow^{\forall} \varphi'$ then $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$.*

Proof (sketch — complete details in [14]). Unfortunately, due to Circularity, a simple induction on the proof tree does not work. Instead, we prove a more general result (Lemma 1 below) allowing sequents with nonempty $\mathcal{A}$ and C, which requires stating semantic assumptions about the rules in $\mathcal{A}$ and C.

First we need to define a more general satisfaction relation than $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$. Let $\delta \in \{+, *\}$ be a flag and let $n \in \mathbb{N}$ be a natural number. We define a new satisfaction relation $\mathcal{S} \models_n^{\delta} \varphi \Rightarrow^{\forall} \varphi'$ by restricting the paths in the definition of $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$ to length at most n, and requiring progress (at least one step) when $\delta = +$.

$$\begin{aligned} \mathit{step}(c,\ c') &\equiv \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists \mathit{FreeVars}(\mu)\ (\varphi_l[c/\square] \wedge \varphi_r[c'/\square]) \\ \mathit{coreach}(\varphi) &\equiv \forall n \forall c_0 ... c_n \left(\square = c_0 \to \bigwedge_{0 \le i < n} \mathit{step}(c_i,\ c_{i+1}) \to \neg \exists c_{n+1}\ \mathit{step}(c_n,\ c_{n+1}) \to \bigvee_{0 \le i \le n} \varphi[c_i/\square] \right) \end{aligned}$$

Fig. 2. FOL encoding of one step transition relation and all-path reachability.

Formally, we define $\mathcal{S} \models_n^\delta \varphi \Rightarrow^\forall \varphi'$ to hold iff for any complete path $\tau = \gamma_1 ... \gamma_k$ of length $k \le n$ and for any ρ such that $(\gamma_1, \rho) \models \varphi$, there exists $i \in \{1, ..., k\}$ such that $(\gamma_i, \rho) \models \varphi'$. Additionally, when $\delta = +$, we require that $i \neq 1$ (i.e. γ makes progress). The indexing on n is required to prove the soundness of circularities. Now we can state the soundness lemma.

Lemma 1. *If $\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^\forall \varphi'$ and $\mathcal{S} \models_n^+ \mathcal{A}$ and $\mathcal{S} \models_{n-1}^+ C$ then $\mathcal{S} \models_n^* \varphi \Rightarrow^\forall \varphi'$, and furthermore, if C is nonempty then $\mathcal{S} \models_n^+ \varphi \Rightarrow^\forall \varphi'$.*

Theorem 1 follows by showing that $\mathcal{S} \models \varphi \Rightarrow^\forall \varphi'$ iff $\mathcal{S} \models_n \varphi \Rightarrow^\forall \varphi'$ for all $n \in \mathbb{N}$.

Lemma 1 is proved by induction on the derivation (with each induction hypothesis universally quantified over n). Consequence, Case Analysis, and Abstraction are easy. Axiom may only be used in cases where $\mathcal{S} \models_n^+ \mathcal{A}$ includes $\mathcal{S} \models_n^+ \varphi \Rightarrow^\forall \varphi'$ (as $\mathcal{S}$ contains only one-path rules). Reflexivity may only be used when C is empty, and $\mathcal{S} \models^* \varphi \Rightarrow^\forall \varphi$ unconditionally. The premises of Step are pattern implications which imply that any configuration matching φ is not stuck in $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$, and all of its immediate successors satisfy φ'. This directly establishes that $\mathcal{S} \models^+ \varphi \Rightarrow^\forall \varphi'$. Transitivity requires considering execution paths more carefully. If C is empty, then the proof is trivial. Otherwise the induction hypothesis gives that $\varphi_1 \Rightarrow^\forall \varphi_2$ holds with progress. Therefore, when proving $\varphi_2 \Rightarrow^\forall \varphi_3$, the circularities are enabled soundly. Circularity proceeds by an inner well-founded induction on n. The outer induction over the derivation gives an induction hypothesis showing the desired conclusion under the additional assumption that $\varphi \Rightarrow^\forall \varphi'$ holds for any m strictly less than n, which is exactly the induction hypothesis provided by the inner induction on n. □

We next show relative completeness: any valid all-path reachability property of any program in any language with an operational semantics given as a reachability system $\mathcal{S}$ is derivable with the proof system in Fig. 1 from $\mathcal{S}$. As with Hoare and dynamic logics, "relative" means we assume an oracle capable of establishing validity in the first-order theory of the state, which here is the configuration model $\mathcal{T}$. An immediate consequence of relative completeness is that Circularity is sufficient to derive any repetitive behavior occurring in any program written in any language, and that Step is also sufficient to derive any non-deterministic behavior! We establish the relative completeness under the following assumptions: (1) $\mathcal{S}$ is finite; (2) the model $\mathcal{T}$ includes natural numbers with addition and multiplication; and (3) the set of configurations $\mathcal{T}_{Cfg}$ is countable (the model $\mathcal{T}$ includes some injective function $\alpha : \mathcal{T}_{Cfg} \to \mathbb{N}$). Assumption (1) ensures Step has a finite number of prerequisites. Assumption (2) is a standard assumption (also made by Hoare and dynamic logic completeness results) which allows the definition of Gödel's β predicate. Assumption (3) allows the encoding of a sequence of configurations into a

IMP language syntax	IMP evaluation contexts syntax
$PVar ::=$ program variables	$Context ::= \blacksquare$
$Exp ::= PVar \mid Int \mid Exp\ \mathtt{op}\ Exp$	$\mid \langle Context,\ State\rangle$
$Stmt ::= \mathtt{skip} \mid PVar := Exp$	$\mid Context\ \mathtt{op}\ Exp \mid Int\ \mathtt{op}\ Context$
$\mid Stmt; Stmt \mid Stmt \parallel Stmt$	$\mid PVar := Context \mid Context; Stmt$
$\mid \mathtt{if}(Exp)\ Stmt\ \mathtt{else}\ Stmt$	$\mid Context \parallel Stmt \mid Stmt \parallel Context$
$\mid \mathtt{while}(Exp)\ Stmt$	$\mid \mathtt{if}(Context)\ Stmt\ \mathtt{else}\ Stmt$

IMP operational semantics			
lookup	$\langle C,\ \sigma\rangle[x] \Rightarrow^{\exists} \langle C,\ \sigma\rangle[\sigma(x)]$	**op**	$i_1\, \mathtt{op}\, i_2 \Rightarrow^{\exists} i_1\ op_{Int}\ i_2$
asgn	$\langle C,\ \sigma\rangle[x := i] \Rightarrow^{\exists} \langle C,\ \sigma[x \leftarrow i]\rangle[\mathtt{skip}]$	**seq**	$\mathtt{skip}; s \Rightarrow^{\exists} s$
cond$_1$	$\mathtt{if}(i)\ s_1\ \mathtt{else}\ s_2 \Rightarrow^{\exists} s_1 \quad$ if $i \neq 0$	**cond**$_2$	$\mathtt{if}(0)\ s_1\ \mathtt{else}\ s_2 \Rightarrow^{\exists} s_2$
while	$\mathtt{while}(e)\ s \Rightarrow^{\exists} \mathtt{if}(e)\ s; \mathtt{while}(e)\ s\ \mathtt{else}\ \mathtt{skip}$	**finish**	$\mathtt{skip} \parallel \mathtt{skip} \Rightarrow^{\exists} \mathtt{skip}$

Fig. 3. IMP language syntax and operational semantics based on evaluation contexts.

sequence of natural numbers. We expect the operational semantics of any reasonable language to satisfy these conditions. Formally, we have the following

Theorem 2 (Relative Completeness). *If $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$ then $\mathcal{S} \vdash \varphi \Rightarrow^{\forall} \varphi'$, for any semantics $\mathcal{S}$ satisfying the three assumptions above.*

Proof (sketch — complete details in [14]). Our proof relies on the fact that pattern reasoning in first-order matching logic reduces to FOL reasoning in the model $\mathcal{T}$. A key component of the proof is defining the $coreach(\varphi)$ predicate in plain FOL. This predicate holds when every complete $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path τ starting at c includes some configuration satisfying φ. We express $coreach(\varphi)$ using auxiliary predicate $step(c,\ c')$ which encodes the one step transition relation ($\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$). Fig. 2 shows both definitions. As it is, $coreach(\varphi)$ is not a proper FOL formula, as it quantifies over a sequence of configurations. This is addressed using the injective function α to encode universal quantification over a sequence of configurations into universal quantification over a sequence of integers, which is in turn encoded into quantification over two integer variables using Gödel's β predicate (encoding shown in [14]).

Next, using the definition above we encode the semantic validity of an all-path reachability rule as FOL validity: $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$ iff $\models \varphi \rightarrow coreach(\varphi')$. Therefore, the theorem follows by Consequence from the sequent $\mathcal{S} \vdash coreach(\varphi') \Rightarrow^{\forall} \varphi'$. We derive this sequent by using Circularity to add the rule to the set of circularities, then by using Step to derive one $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-step, and then by using Transitivity and Axiom with the rule itself to derive the remaining $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-steps (circularities can be used after Transitivity). The formal derivation uses all eight proof rules. □

6 Verifying Programs

In this section we show a few examples of using our proof system to verify programs based on an operational semantics. In a nutshell, the proof system enables generic symbolic execution combined with circular reasoning. Symbolic execution is achieved by rewriting modulo domain reasoning.

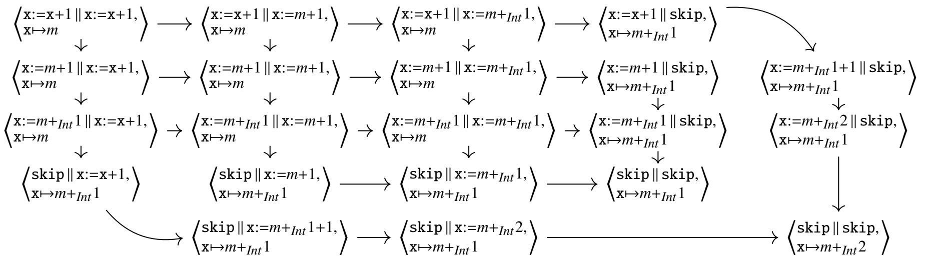
⟨x:=x+1 ∥ x:=x+1, x↦m⟩
⟨x:=x+1 ∥ x:=m+1, x↦m⟩
⟨x:=x+1 ∥ x:=m+Int 1, x↦m⟩
⟨x:=x+1 ∥ skip, x↦m+Int 1⟩
⟨x:=m+1 ∥ x:=x+1, x↦m⟩
⟨x:=m+1 ∥ x:=m+1, x↦m⟩
⟨x:=m+1 ∥ x:=m+Int 1, x↦m⟩
⟨x:=m+1 ∥ skip, x↦m+Int 1⟩
⟨x:=m+Int 1+1 ∥ skip, x↦m+Int 1⟩
⟨x:=m+Int 1 ∥ x:=x+1, x↦m⟩
⟨x:=m+Int 1 ∥ x:=m+1, x↦m⟩
⟨x:=m+Int 1 ∥ x:=m+Int 1, x↦m⟩
⟨x:=m+Int 1 ∥ skip, x↦m+Int 1⟩
⟨x:=m+Int 2 ∥ skip, x↦m+Int 1⟩
⟨skip ∥ x:=x+1, x↦m+Int 1⟩
⟨skip ∥ x:=m+1, x↦m+Int 1⟩
⟨skip ∥ x:=m+Int 1, x↦m+Int 1⟩
⟨skip ∥ skip, x↦m+Int 1⟩
⟨skip ∥ x:=m+Int 1+1, x↦m+Int 1⟩
⟨skip ∥ x:=m+Int 2, x↦m+Int 1⟩
⟨skip ∥ skip, x↦m+Int 2⟩

Fig. 4. State space of the racing increment example

First, we introduce a simple parallel imperative language, IMP. Fig. 3 shows its syntax and an operational semantics based on evaluation contexts [11] (we choose evaluation contexts for presentation purposes only). IMP has only integer expressions. When used as conditions of `if` and `while`, zero means false and any non-zero integer means true (like in C). Expressions are formed with integer constants, program variables, and conventional arithmetic constructs. Arithmetic operations are generically described as `op`. IMP statements are assignment, `if`, `while`, sequential composition and parallel composition. IMP has shared memory parallelism without explicit synchronization. The examples use the parallel construct only at the top-level of the programs. The second example shows how to achieve synchronization using the existing language constructs.

The program configurations of IMP are pairs $\langle \texttt{code},\ \sigma\rangle$, where `code` is a program fragment and σ is a state term mapping program variables into integers. As usual, we assume appropriate definitions for the integer and map domains available, together with associated operations like arithmetic operations ($i_1\ op_{Int}\ i_2$, etc.) on the integers and lookup ($\sigma(x)$) and update ($\sigma[x \leftarrow i]$) on the maps. We also assume a context domain with a plugging operation ($C[t]$) that composes a context and term back into a configuration. A configuration context consists of a code context and a state. The definition in Fig. 3 consists of eight reduction rules between program configurations, which make use of first-order variables: x is a variable of sort *PVar*; e is a variable of sort *Exp*; s, s_1, s_2 are variables of sort *Stmt*; i, i_1, i_2 are variables of sort *Int*; σ is a variable of sort *State*; C is a variable of sort *Context*. A rule reduces a configuration by splitting it into a context and a redex, rewriting the redex and possibly the context, and then plugging the resulting term into the resulting context. As an abbreviation, a context is not mentioned if not used; e.g., the rule **op** is in full $\langle C,\ \sigma\rangle[i_1\ \texttt{op}\ i_2] \Rightarrow^{\exists} \langle C,\ \sigma\rangle[i_1\ op_{Int}\ i_2]$. For example, configuration $\langle \texttt{x} := (2 + 5) - 4,\ \sigma\rangle$ reduces to $\langle \texttt{x} := 7 - 4,\ \sigma\rangle$ by applying the $\mathbf{op}_+$ rule with $C \equiv \texttt{x} := \blacksquare - 4$, $\sigma \equiv \sigma$, $i_1 \equiv 2$ and $i_2 \equiv 5$. We can regard the operational semantics of IMP above as a set of reduction rules of the form "$l \Rightarrow^{\exists} r$ if b", where l and r are program configurations with variables constrained by boolean condition b. As discussed in Section 3, our proof system works with any rules of this form.

Next, we illustrate the proof system on a few examples. The first example shows that our proof system enables exhaustive state exploration, similar to symbolic model-checking but based on the operational semantics. Although humans prefer to avoid such explicit proofs and instead methodologically use abstraction or compositional reasoning whenever possible (and such methodologies are not excluded by our proof system), a complete proof system must nevertheless support them. The code $\mathtt{x{:=}x{+}1} \parallel \mathtt{x{:=}x{+}1}$ exhibits a race on $\mathtt{x}$: the value of $\mathtt{x}$ increases by 1 when both reads happen before either write, and by 2 otherwise. The all-path rule that captures this behavior is

$$\langle \mathtt{x{:=}x{+}1} \parallel \mathtt{x{:=}x{+}1},\ \mathtt{x}{\mapsto}m\rangle \Rightarrow^{\forall} \exists n\,(\langle \mathtt{skip},\ \mathtt{x}{\mapsto}n\rangle \wedge (n = m +_{Int} 1 \vee n = m +_{Int} 2)$$

We show that the program has exactly these behaviors by deriving this rule in the proof system. Call the right-hand-side pattern G. The proof contains subproofs of $c \Rightarrow^{\forall} G$ for every reachable configuration c, tabulated in Fig. 4. The subproofs for c matching G use Reflexivity and Consequence, while the rest use Transitivity, Step, and Case Analysis to reduce to the proofs for the next configurations. For example, the proof fragment below shows how $\langle \mathtt{x} := m + 1 \parallel \mathtt{x} := \mathtt{x} + 1,\ \mathtt{x} \mapsto m\rangle \Rightarrow^{\forall} G$ reduces to $\langle \mathtt{x} := m +_{Int} 1 \parallel \mathtt{x} := \mathtt{x}{+}1,\ \mathtt{x} \mapsto m\rangle \Rightarrow^{\forall} G$ and $\langle \mathtt{x} := m{+}1 \parallel \mathtt{x} := m{+}1,\ \mathtt{x} \mapsto m\rangle \Rightarrow^{\forall} G$:

$$\dfrac{\textsc{Step}\ \dfrac{\cdots}{\left\langle\begin{smallmatrix}\mathtt{x{:=}}m\mathtt{+1} \parallel \mathtt{x{:=}x{+}1,}\\ \mathtt{x}{\mapsto}m\end{smallmatrix}\right\rangle \Rightarrow^{\forall} \begin{array}{l}\left\langle\begin{smallmatrix}\mathtt{x{:=}}m{+_{Int}}1 \parallel \mathtt{x{:=}x{+}1,}\\ \mathtt{x}{\mapsto}m\end{smallmatrix}\right\rangle\\ \vee \left\langle\begin{smallmatrix}\mathtt{x{:=}}m{+}1 \parallel \mathtt{x{:=}}m{+}1,\\ \mathtt{x}{\mapsto}m\end{smallmatrix}\right\rangle\end{array}} \quad \dfrac{\dfrac{\cdots}{\left\langle\begin{smallmatrix}\mathtt{x{:=}}m{+_{Int}}1 \parallel \mathtt{x{:=}x{+}1,}\\ \mathtt{x}{\mapsto}m\end{smallmatrix}\right\rangle \Rightarrow^{\forall} G} \ \dfrac{\cdots}{\left\langle\begin{smallmatrix}\mathtt{x{:=}}m{+}1 \parallel \mathtt{x{:=}}m{+}1,\\ \mathtt{x}{\mapsto}m\end{smallmatrix}\right\rangle \Rightarrow^{\forall} G}}{\left\langle\begin{smallmatrix}\mathtt{x{:=}}m{+_{Int}}1 \parallel \mathtt{x{:=}x{+}1,}\\ \mathtt{x}{\mapsto}m\end{smallmatrix}\right\rangle \vee \left\langle\begin{smallmatrix}\mathtt{x{:=}}m{+}1 \parallel \mathtt{x{:=}}m{+}1,\\ \mathtt{x}{\mapsto}m\end{smallmatrix}\right\rangle \Rightarrow^{\forall} G}\ \text{CA}}{\langle \mathtt{x} := m + 1 \parallel \mathtt{x} := \mathtt{x} + 1,\ \mathtt{x} \mapsto m\rangle \Rightarrow^{\forall} G}\ \textsc{Trans}$$

For the rule hypotheses of Step above, note that all rules but **lookup** and **op**$_+$ make the overlap condition $\exists c\ \left(\left\langle\begin{smallmatrix}\mathtt{x{:=}}m\mathtt{+1} \parallel \mathtt{x{:=}x{+}1,}\\ \mathtt{x}{\mapsto}m\end{smallmatrix}\right\rangle[c/\square] \wedge \varphi_l[c/\square]\right)$ unsatisfiable, and only one choice of free variables works for the **lookup** and **op**$_+$ rules. For **lookup**, φ_l is $\langle C, \sigma\rangle[x]$ and the overlap condition is only satisfiable if the logical variables C, σ and x are equal to $(\mathtt{x} := m + 1 \parallel \mathtt{x} := \blacksquare + 1)$, $(\mathtt{x} \mapsto m)$, and $\mathtt{x}$, resp. Under this assignment, the pattern $\varphi_r = \langle C, \sigma\rangle[\sigma(x)]$ is equivalent to $\langle \mathtt{x} := m + 1 \parallel \mathtt{x} := m + 1,\ \mathtt{x} \mapsto m\rangle$, the right branch of the disjunction. The **op**$_+$ rule is handled similarly. The assignment for **lookup** can also witness the existential in the progress hypothesis of Step. Subproofs for other states in Fig. 4 can be constructed similarly.

The next two examples use loops and thus need to state and prove invariants. As discussed in [4], Circularity generalizes the various language-specific invariant proof rules encountered in Hoare logics. One application is reducing a proof of $\varphi \Rightarrow^{\forall} \varphi'$ to proving $\varphi_{\text{inv}} \Rightarrow^{\forall} \varphi_{\text{inv}} \vee \varphi'$ for some pattern invariant φ_{inv}. We first show $\models \varphi \to \varphi_{\text{inv}}$, and use Consequence to change the goal to $\varphi_{\text{inv}} \Rightarrow^{\forall} \varphi'$. This is claimed as a circularity, and then proved by transitivity with $\varphi_{\text{inv}} \vee \varphi'$. The second hypothesis $\{\varphi_{\text{inv}} \Rightarrow^{\forall} \varphi'\} \vdash \varphi_{\text{inv}} \vee \varphi' \Rightarrow^{\forall} \varphi'$ is proved by Case Analysis, Axiom, and Reflexivity.

```
f0 = 1;                  f1 = 1;
turn = 1;                turn = 0;
while (f1 && turn)       while (f0 && (1 - turn))
  skip                     skip
x = x + 1;               x = x + 1;
f0 = 0;                  f1 = 0;
```

Fig. 5. Peterson's algorithm (threads $\mathtt{T}_0$ and $\mathtt{T}_1$)

Next, we can use Peterson's algorithm for mutual exclusion to eliminate the race as shown in Fig. 5. The all-path rule $\varphi \Rightarrow^{\forall} \varphi'$ that captures the new behavior is

$$\begin{array}{l}\langle \mathtt{T}_0 \parallel \mathtt{T}_1,\ (\mathtt{f0} \mapsto 0,\ \mathtt{f1} \mapsto 0,\ \mathtt{x} \mapsto N)\rangle\\ \Rightarrow^{\forall} \exists t\,\langle \mathtt{skip},\ (\mathtt{f0} \mapsto 0,\ \mathtt{f1} \mapsto 0,\ \mathtt{x} \mapsto N +_{Int} 2,\ \mathtt{turn} \mapsto t)\rangle\end{array}$$

Similarly to the unsynchronized example, the proof contains subproofs of $c \Rightarrow^{\forall} \varphi'$ for every configuration c reachable from φ. The main difference is that Circularity is used with each of these rules $c \Rightarrow^{\forall} \varphi'$ with one of the two threads of c in the `while` loop (these rules capture the invariants). Thus, when we reach a configuration c visited before, we use the rule added by Circularity to complete the proof.

The final example is the program `SUM` ≡ "`s := 0; LOOP`" (where `LOOP` stands for "`while (n>0) (s := s+n; n := n-1)`"), which computes in `s` the sum of the numbers from 1 up to `n`. The all-path reachability rule $\varphi \Rightarrow^{\forall} \varphi'$ capturing this behavior is

$$\langle \texttt{SUM},\ (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)\rangle \land n \geq_{Int} 0 \ \Rightarrow^{\forall}\ \langle \texttt{skip},\ (\texttt{s} \mapsto n *_{Int} (n +_{Int} 1)/_{Int} 2,\ \texttt{n} \mapsto 0)\rangle$$

We derive the above rule in our proof system by using Circularity with the invariant rule $\exists n'(\langle \texttt{LOOP},\ (\texttt{s}\mapsto(n-_{Int}n')*_{Int}(n+_{Int}n'+_{Int}1)/_{Int}2,\ \texttt{n}\mapsto n')\rangle \land n' \geq_{Int} 0) \Rightarrow^{\forall} \varphi'$. Previous work [4–7] presented a proof system able to derive similar rules, but which hold along *some* execution path, requiring a separate proof that the program is deterministic.

7 Implementation

Here we briefly discuss our prototype implementation of the proof system in Fig. 1 in $\mathbb{K}$ [9]. We choose $\mathbb{K}$ because it is a modular semantic language design framework, it is used for teaching programming languages at several universities, and there are several languages defined in it including C [10], PHP [15], Python, and Java. Due to space limitations, we do not present $\mathbb{K}$ here. We refer the reader to `http://kframework.org` for language definitions, a tutorial, and our prototype. As discussed in Section 3, we simply view a $\mathbb{K}$ semantics as a set of reachability rules of the form "$l \land b \Rightarrow^{\exists} r$".

The prototype is implemented in Java, and uses Z3 [16] for domain reasoning. It takes an operational semantics and uses it to perform concrete or symbolic execution. At its core, it performs *narrowing* of a conjunctive pattern with reachability rules between conjunctive patterns, where a conjunctive pattern is a pattern of the form $\exists X(\pi \land \psi)$, with X a set of variables, π a basic pattern (program configurations with variables), and ψ a structureless formula. Narrowing is necessary when a conjunctive pattern is too abstract to match the left-hand side of any rule, but is unifiable with the left-hand sides of some rules. For instance, consider the IMP code fragment "`if (`b`) then x = 1; else x = 0;`". This code does not match the left-hand sides of either of the two rules giving semantics to `if` (similar to **cond**$_1$ and **cond**$_2$ in Fig. 3), but it is unifiable with the left-hand sides of both rules. Intuitively, if we use the rules of the semantics, taking steps of rewriting on a ground configuration yields concrete execution, while taking steps of narrowing on a conjunctive pattern yields symbolic execution. In our practical evaluation, we found that conjunctive patterns tend to suffice to specify both the rules for operational semantics and program specifications.

For each step of narrowing, the $\mathbb{K}$ engine uses *unification modulo theories*. In our implementation, we distinguish a number of mathematical theories (e.g. booleans, integers, sequences, sets, maps, etc) which the underlying SMT solver can reason about. Specifically, when unifying a conjunctive pattern $\exists X(\pi \land \psi)$ with the left-hand side of a rule $\exists X_l(\pi_l \land \psi_l)$ (we assume $X \cap X_l = \emptyset$), the $\mathbb{K}$ engine begins with the syntactic unification of the basic patterns π and π_l. Upon encountering corresponding subterms (π' in π and π'_l in π_l) which are both terms of one of the theories above, it records an equality $\pi' = \pi'_l$ rather than decomposing the subterms further (if one is in a theory, and the other one is in a different theory or is not in any theory, the unification fails). If this stage of unification is successful, we end up with a conjunction ψ_u of constraints, some having a variable in one side and some with both sides in one of the theories. Satisfiability of $\exists X \cup X_l(\psi \wedge \psi_u \wedge \psi_l)$ is then checked by the SMT solver. If it is satisfiable, then narrowing takes a step from $\exists X(\pi \wedge \psi)$ to $\exists X \cup X_l \cup X_r(\pi_r \wedge \psi \wedge \psi_u \wedge \psi_l \wedge \psi_r)$, where $\exists X_r(\pi_r \wedge \psi_r)$ is the right-hand side of the rule. Intuitively, "collecting" the constraints $\psi_u \wedge \psi_l \wedge \psi_r$ is similar to collecting the path constraint in traditional symbolic execution (but is done in a language-generic manner). For instance, in the `if` case above, narrowing with the two semantics rules results in collecting the constraints $b = \texttt{true}$ and $b = \texttt{false}$.

The $\mathbb{K}$ engine accepts a set of user provided rules to prove together, which capture the behavior of the code being verified. Typically, these rules specify the behavior of recursive functions and while loops. For each rule, the $\mathbb{K}$ engine searches starting from the left-hand side for formulae which imply the right-hand side, starting with $\mathcal{S}$ the semantics and C all the rules it attempts to prove. By a derived rule called *Set Circularity*, this suffices to show that each rule is valid. As an optimization, Axiom is given priority over Step (use specifications rather than stepping into the code).

Most work goes into implementing the Step proof rule, and in particular calculating how $\rho \models \exists c\ (\varphi[c/\square] \wedge \varphi_l[c/\square])$ can be satisfied. This holds when $\rho^\gamma \models \varphi$ and $\rho^\gamma \models \varphi_l$, which can be checked with unification modulo theories. To use Step in an automated way, the $\mathbb{K}$ tool constructs φ' for a given φ as a disjunction of $\varphi_r \wedge \psi_u \wedge \psi \wedge \psi_l$ over each rule $\varphi_l \Rightarrow^\exists \varphi_r \in \mathcal{S}$ and each way ψ_u of unifying φ with φ_l. As discussed in Section 4, in general this disjunction may not be finite, but it is sufficient for the examples that we considered. The Consequence proof rule also requires unification modulo theories, to check validity of the implication hypothesis $\models \varphi_1 \rightarrow \varphi'_1$. The main difference from Step is that the free variables of φ' become universality quantified when sending the query to the SMT solver. The implementation of the other proof rules is straight-forward.

8 Related Work

Using Hoare logic [17] to prove concurrent programs correct dates back to Owicki and Gries [18]. In the rely-guarantee method proposed by Jones [19] each thread relies on some properties being satisfied by the other threads, and in its turn, offers some guarantees on which the other threads can rely. O'Hearn [20] advances a Separation Hypothesis in the context of separation logic [21] to achieve compositionality: the state can be partitioned into separate portions for each process and relevant resources, respectively, satisfying certain invariants. More recent research focuses on improvements over both of the above methods and even combinations of them (e.g., [22–25]).

The satisfaction of all-path-reachability rules can also be understood intuitively in the context of temporal logics. Matching logic formulae can be thought of as state formulae, and reachability rules as temporal formulae. Assuming CTL* on finite traces, the semantics rule $\varphi \Rightarrow^\exists \varphi'$ can be expressed as $\varphi \rightarrow E\bigcirc\varphi'$, while an all-path reachability rule $\varphi \Rightarrow^\forall \varphi'$ can be expressed as $\varphi \rightarrow A\Diamond\varphi'$. However, unlike in CTL*, the φ and φ' formulae of reachability rules $\varphi \Rightarrow^\exists \varphi'$ or $\varphi \Rightarrow^\forall \varphi'$ share their free variables. Thus, existing proof systems for temporal logics (e.g., the CTL* one by Pnueli and Kesten) are not directly comparable with our approach.

Bae et al [26], Rocha and Meseguer [27], and Rocha et al [28] use narrowing to perform symbolic reachability analysis in a transition system associated to a unconditional rewrite theory for the purposes of verification. There are two main differences between their work and ours. First, they express state predicates in equational theories. Matching logic is more general, being first-order logic over a model of configurations T. Consequently, the Step proof rule takes these issues into account when considering the successors of a state. Second, they use rewrite systems for symbolic model checking. Our work is complementary, in the sense that we use the operational semantics for program verification, and check properties more similar to those in Hoare logic.

Language-independent proof systems A first proof system is introduced in [6], while [5] presents a mechanical translation from Hoare logic proof derivations for IMP into derivations in the proof system. The Circularity proof rule is introduced in [4]. Finally, [7] supports operational semantics given with conditional rules, like small-step and big-step. All these previous results can only be applied to deterministic programs.

9 Conclusion and Future Work

This paper introduces a sound and (relatively) complete language-independent proof system which derives program properties holding along all execution paths (capturing partial correctness for non-deterministic programs), directly from an operational semantics. The proof system separates reasoning about deterministic language features (via the operational semantics) from reasoning about non-determinism (via the proof system). Thus, all we need in order to verify programs in a language is an operational semantics for the respective language.

We believe that existing techniques such as rely-guarantee and concurrent separation logic could be used in conjunction with our proof system to achieve semantically grounded and compositional verification.

Our approach handles operational semantics given with unconditional rules, like $\mathbb{K}$ framework, PLT-Redex, and CHAM, but it cannot handle operational semantics given with conditional rules, like big-step and small-step (rules with premises). Extending the presented results to work with conditional rules would boil down to extending the Step proof rule, which derives the fact that φ reaches φ' in one step along all execution paths. Such a extended Step would have as prerequisites whether the left-hand side of a semantics rule matches (like the existing Step) and additionally whether its premises hold. The second part would require an encoding of reachability in first-order logic, which is non-trivial and mostly likely would result in a first-order logic over a richer model than $\mathcal{T}$. The difficulty arises from the fact that Step must ensure all successors of φ are in φ'. Thus, this extension is left as future work.

Acknowledgements We would like to thank the anonymous reviewers and the FSL members for their valuable feedback on an early version of this paper. The work presented in this paper was supported in part by the Boeing grant on "Formal Analysis Tools for Cyber Security" 2014-2015, the NSF grant CCF-1218605, the NSA grant H98230-10-C-0294, the DARPA HACMS program as SRI subcontract 19-000222, the Rockwell Collins contract 4504813093, and the (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010.

References

1. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics textbook. Formal Aspects of Computing **10** (1998) 171–186

2. Jacobs, B.: Weakest pre-condition reasoning for Java programs with JML annotations. J. Logic and Algebraic Programming **58**(1-2) (2004) 61–88
3. Appel, A.W.: Verified software toolchain. In: ESOP. Volume 6602 of LNCS. (2011) 1–17
4. Roșu, G., Ștefănescu, A.: Checking reachability using matching logic. In: OOPSLA, ACM (2012) 555–574
5. Roșu, G., Ștefănescu, A.: From Hoare logic to matching logic reachability. In: FM. Volume 7436 of LNCS. (2012) 387–402
6. Roșu, G., Ștefănescu, A.: Towards a unified theory of operational and axiomatic semantics. In: ICALP. Volume 7392 of LNCS. (2012) 351–363
7. Roşu, G., Ştefănescu, A., Ciobâcă, c., Moore, B.M.: One-path reachability logic. In: LICS'13, IEEE (2013)
8. Roșu, G., Ellison, C., Schulte, W.: Matching logic: An alternative to Hoare/Floyd logic. In: AMAST. Volume 6486 of LNCS. (2010) 142–162
9. Roșu, G., Șerbănuță, T.F.: An overview of the K semantic framework. J. Logic and Algebraic Programming **79**(6) (2010) 397–434
10. Ellison, C., Roșu, G.: An executable formal semantics of C with applications. In: POPL, ACM (2012) 533–544
11. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT (2009)
12. Berry, G., Boudol, G.: The chemical abstract machine. Theoretical Computer Science **96**(1) (1992) 217–248
13. Matthews, J., Findler, R.B.: An operational semantics for Scheme. JFP **18**(1) (2008) 47–86
14. Ştefănescu, A., Ciobâcă, c., Moore, B.M., Şerbănuţă, T.F., Roşu, G.: Reachability Logic in K. Technical Report http://hdl.handle.net/2142/46296, University of Illinois (Nov 2013)
15. Filaretti, D., Maffeis, S.: An executable formal semantics of php. In: ECOOP. LNCS (2014) to appear.
16. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. Volume 4963 of LNCS. (2008)
17. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1969) 576–580
18. Owicki, S.S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. Communications of the ACM **19**(5) (1976) 279–285
19. Jones, C.B.: Specification and design of (parallel) programs. In Mason, R.E.A., ed.: Information Processing 1983: World Congress Proceedings, Elsevier (1984) 321–332
20. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theoretical Computer Science **375**(1-3) (2007) 271–307
21. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, IEEE (2002) 55–74
22. Feng, X.: Local rely-guarantee reasoning. In: POPL, ACM (2009) 315–327
23. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR. Volume 4703 of LNCS. (2007) 256–271
24. Reddy, U.S., Reynolds, J.C.: Syntactic control of interference for separation logic. In: POPL, ACM (2012) 323–336
25. Hayman, J.: Granularity and concurrent separation logic. In: CONCUR. Volume 6901 of LNCS. (2011) 219–234
26. Bae, K., Escobar, S., Meseguer, J.: Abstract logical model checking of infinite-state systems using narrowing. In: RTA. (2013) 81–96
27. Rocha, C., Meseguer, J.: Proving safety properties of rewrite theories. In: CALCO. Volume 6859 of LNCS. (2011) 314–328
28. Rocha, C., Meseguer, J., Muñoz, C.A.: Rewriting modulo smt and open system analysis. In: WRLA. LNCS (2014) to appear.